Universidade do Minho
Escola de Engenharia

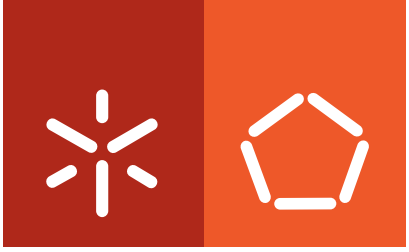
Victor Francisco Mendes de Freitas Gomes da Fonte

Causality Tracking in Dynamic Distributed Systems

Outubro de 2008

Victor Francisco Mendes de Freitas Gomes da Fonte **Causality Tracking in Dynamic Distributed Systems**

UMinho | 2008



Universidade do Minho
Escola de Engenharia

Victor Francisco Mendes de Freitas
Gomes da Fonte

Causality Tracking in Dynamic Distributed Systems

Tese de Doutoramento
Tecnologias da Programação

Trabalho efectuado sob a orientação do
Professor Doutor Carlos Baquero Moreno
e do
Professor Doutor Paulo Sérgio Almeida

Outubro de 2008

DECLARAÇÃO

Victor Francisco Mendes de Freitas Gomes da Fonte

Endereço electrónico: **vff@di.uminho.pt**

Telefone: **253604449**

Título da dissertação/tese:

Causality Tracking in Dynamic Distributed Systems

Orientador:

Professor Doutor Carlos Baquero Moreno
Professor Doutor Paulo Sérgio Almeida

Ano de Conclusão: **2008**

Designação do Ramo de Conhecimento do Doutoramento:

Tecnologias da Programação

Universidade do Minho, Outubro de 2008

Assinatura: _____

Resumo

A causalidade desempenha um papel central no tratamento de problemas importantes de sistemas distribuídos, tais como na replicação de dados, na análise de execuções, na comunicação em grupo e na determinação de estados globais. Por forma a ser útil, a causalidade precisa de ser concretizada em mecanismos que procedam ao seu registo.

Os mecanismos existentes, tais como os vectores versão e os relógios vectoriais, assumem a existência de um mapeamento entre identificadores globalmente únicos e contadores inteiros. Num sistema em que é conhecido o número de entidades, é possível pré-configurar estes identificadores por forma a ocuparem posições distintas num vector ou serem-lhe atribuídos nomes distintos. A gestão destas identidades é bem mais problemática em ambientes dinâmicos, com grande número de entidades e onde estas são permanentemente criadas e destruídas. Esta situação é agravada na presença de partições de rede. As soluções actuais para o registo de causalidade não se revelam apropriadas a estes cenários, cada vez mais relevantes.

Esta tese apresenta novos mecanismos de registo de causalidade que têm a propriedade de poder ser usados em cenários com um número dinâmico de entidades. Estes mecanismos permitem a criação descentralizada de entidades (processos ou réplicas) sem requerer identificadores globais ou coordenação global para a sua geração. Estes mecanismos apresentam codificações com tamanho variável, o que permite uma adaptação automática ao número de entidades em jogo, crescendo e colapsando de acordo com as necessidades.

Abstract

Causality plays a central role as a building block in solving important problems in distributed systems, such as replication, debugging, group communication and global snapshots. To be useful, causality must be realised by actual mechanisms that can track it and encode it.

Existing causality tracking mechanisms, such as vector clocks and version vectors, rely on mappings from globally unique identifiers to integer counters. In a system with a well known set of entities these identifiers can be pre-configured and given distinct positions in a vector or distinct names in a mapping. Identity management is more problematic in dynamic systems, with a large and highly variable number of entities, being worsened when network partitions occur. Present solutions for causality tracking are not appropriate to these increasingly common scenarios.

This thesis introduces novel causality tracking mechanisms that can be used in scenarios with a dynamic number of entities. These allow completely decentralised creation of entities (processes or replicas) with no need for global identifiers or global coordination. These mechanisms have a variable size representation that adapts automatically to the number of entities, growing or shrinking appropriately.

Acknowledgements

First of all, I want to thank Carlos Baquero and Paulo Sérgio Almeida for accepting being my advisers. Their insightful guidance, their patience and constant support profoundly shaped this work.

For his initial call, that ultimately lured me into the field of Distributed Systems, and above all things, for a friendship that I treasure, I'm deeply grateful to Francisco Moura.

To all past and current members of the Distributed Systems Group I want to thank for making this such a fun and always exciting place to work. Special thanks to Rui, Orlando, Tó, Zé Pedro, Coutinho, and, of course, to Carlos, Paulo Sérgio and Francisco Moura, for their encouragement and all the fruitful discussions.

Finally, I would like to thank my sisters, Ana and Bárbara, for being my biggest fans, and my parents Maria de Jesus and Victor for raising me so wisely and for their unconditional support in all aspects of my life.

To my loving wife, Teresa, I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ordering of events has always been a crucial issue in distributed systems. The knowledge that an event has occurred after another and that it is a potential consequence of the other (i.e. there is a *causal* relationship between them) forms the basis of many algorithms used in distributed applications. Examples include communication protocols, distributed debugging, termination detection, distributed mutual exclusion, data management and version control.

These issues have been studied for many years, since the initial formalisation of the concept in the late 70s [Lam78]. Solutions have been proposed, in the subsequent decade [PPR+83, Fid89, Mat89c], that are easy to implement with a centralised configuration service or when the number of nodes is known in advance. The assumptions made in the initial implementations of the concept (vector clocks and version vectors) fitted the spirit of a time when systems were deployed on LANs, and the scale and dynamics of the internet was much more contained.

Along the last two decades there was a strong push towards more dynamic scenarios. This was clearly driven by technological and theoretical advances that came with the advent of mobile computing [Sat96], peer-to-peer [Ora01], population protocols [AR07], sensor networks [CES04] and more recently cloud computing [Vog08]. In these settings, distributed interactions are no longer among a well known set of entities, either processes or data items. They now include evolving memberships of participating entities in possibly large numbers and high churn

rates, autonomic deployment of new entities, disconnection and re-integration of entities.

1.1 Decentralised Causality Tracking

The abstract definition of causality relies on describing a partial ordering of events or states that are causally related in the course of a distributed computation. Events are related when processes perform computation steps, when messages are sent or received, when data is updated, etc. The transitive closure of this relationship leads to a partial order, which can be represented by a directed acyclic graph.

In its abstract form causality stems naturally from the flow of potential influence that spawns from distributed runs and is independent from whether processes can have a local access to it. However, to be useful, causality must be realised by actual mechanisms that can track it, represent it as actual bits of information, and answer questions such as: "are events a and b concurrent?" or is "a in the causal past of b?"

Version vectors and vector clocks, the classic mechanisms that realise causality and make it usable in actual distributed applications introduce some centralisation assumptions. These are implicit in the use of a vector position for each participating entity or in requiring globally unique names for each entity. This is easily acceptable in systems with static memberships and when no partitions occur. In these contexts it is easy to pre-configure global identities or to run system-wide distributed protocols to obtain a consensus on new identities. However, under high dynamics [Ray06], or when subject to network partitioning, this is no longer an adequate solution. Even if pre-configuration of globally unique identities might seem a feasible option (say relying on MAC addresses) there are still important drawbacks:

- In the presence of churn, either in P2P or cloud computing [DHJ+07] settings, identifiers of entities that once participated in the computation but are no longer active keep polluting the tracking mechanisms. This has lead

to garbage collection approaches but a truly efficient solution is still an open problem.

- The need for globally unique identifiers often implies long identifiers. If millions of sensors are produced with unique identifiers but only an hundred are deployed in a given place and initiate a distributed computation they still incur in the energy drain of transmitting an oversized state.

This dissertation develops a solution to causality tracking in dynamic settings and presents actual mechanisms that can emulate the full functionality of version vectors and vector clocks, while relaxing the existing centralisation assumptions that are rooted on classic identifier requirements. The model is relaxed so that only pair-wise interactions are needed between entities (for any pattern of interactions) and so that any entity can spawn new ones or register the termination of any other entity.

The proposed mechanisms are more complex than the traditional ones but allow a more flexible use. One of these mechanism in particular, Interval Tree Clocks, presented in Chapter 6, even though it can be used in a wider range of scenarios, manages to have a compact representation in the order of magnitude of the classical ones, an important factor for the actual adoption and deployment of the thesis' contributions.

1.2 Dissertation Outline

The contributions in this thesis were published in four peer reviewed articles from 2000 to 2008. The outline of the central chapters (Chapter 3 to Chapter 6) follows this chronology and includes these articles.

Chapter 1 This introduction.

Chapter 2 A review is made on the related work on causality and its tracking mechanisms. Special focus is made on approaches that target more dynamic

settings and that address creation and retirement of counters. This chapter complements the related work already presented in the article chapters, where these approaches are discussed in a more compact and targeted way.

Chapter 3 This chapter presents the Panasync tool, a mechanism for causality tracking that supports ad-hoc file copying, that was published in:

> Paulo Sérgio Almeida, Carlos Baquero and Victor Fonte. *Panasync: dependency tracking among file copies* ACM SIGOPS European Workshop, pp. 7-12, ACM, 2000.

The motivation for Panasync was rooted on the observation that distributed file systems for mobile settings are typically based on volume replication [KS91, GHM+90], where whole sub-directories are kept in synch across a set of machines. In Panasync individual files are the unit of replication. Many replicas of an entity (file) can co-exist in a single machine and different entities can have different numbers of replicas, placed in different devices. All coordination is done locally, in what is now called an autonomic computing approach. In order to track this *data causality* in these dynamic settings, Panasync introduced the first working prototype of *Dynamic Version Stamps.*

Chapter 4 This chapter presents the Dynamic Version Stamps mechanism. A condensed version of the proof was published in

> Paulo Sérgio Almeida, Carlos Baquero and Victor Fonte. *Version Stamps: Decentralised Version Vectors* 22th International Conference on Distributed Computing Systems (ICDCS 2002), pp. 544-551, IEEE, July 2002.

This chapter extends this article by including the full proof of correctness for the mechanism. Dynamic Version Stamps is the first deterministic mechanism that tracks data causality in dynamic systems and introduces a decentralised technique to manage identifiers that mimics some techniques used in termination detection. Interestingly, it can track causality without the

use of counters. The mechanism can substitute the use of version vectors for comparing co-existing entities in a consistent cut, but not vector clocks, since it cannot handle arbitrary causal pasts.

Chapter 5 This chapter presents a description of the state consumption limitations of version stamps under some possible runs, and introduces a synopsis of a new mechanism that can overcome those limitations.

> Paulo Sérgio Almeida, Carlos Baquero and Victor Fonte. *Improving on Version Stamps.* OTM Workshops (2), Lecture Notes in Computer Science, Vol. 4806, pp. 1025-1031, Springer, 2007.

Version Stamps although correctly tracking data causality, exhibit a pathological space growth under some runs, which seriously limits the practical applications of the approach. This aspect is dealt with by considering a mixed approach which combines Dynamic Version Stamps-like identifiers with the use of counters. The resulting mechanism, *Dynamic Map Clocks*, was tested in a reference implementation. While working on the formalisation, a substantial improvement was made, resulting into what became the mechanism presented in the following chapter.

Chapter 6 This chapter presents *Interval Tree Clocks*, a mechanism that can track both data and process causality in dynamic settings, being a generalisation of both version vectors and vector clocks.

> Paulo Sérgio Almeida, Carlos Baquero and Victor Fonte. *Interval Tree Clocks: A Logical Clock for Dynamic Systems. To appear in* OPODIS 2008. Lecture Notes in Computer Science, Vol. 5401, pp. 259-274,Springer 2008.

Apart from the mechanism itself, the chapter presents two contributions that stand on their own: the *Fork-Event-Join* model, a kernel that can be used to unify the modelling of both process and data causality; and a general framework, based on function spaces, to describe clock mechanisms, with invariants towards ensuring correctness of candidate mechanisms. The chapter

also presents simulation-based analysis of space consumption, which shows the space requirements of this mechanism to be modest.

Chapter 7 This chapter discusses the results, summarises the main contributions and points out to some research directions.

Finally, Appendixes A, B and C present reference implementations of the mechanisms introduced in Chapters 3, 5 and 6.

Chapter 2

Logical Clock Systems

The notions of causality and time play a central role in the design of distributed algorithms. Although a global time reference would help solving some problems in the field of distributed systems, this reference is usually unavailable. In spite of that, it is often helpful to know the relative order in which events take place in the systems, and for a number of fundamental applications this is often an actual requirement.

Knowledge of this relative order can be achieved even in totally asynchronous systems that have no way of measuring the passage of real time, just by observing the causal precedence relation between events. This chapter starts by introducing the notion of causality and its application in distributed systems. It follows by its formal definition, and by introducing the happens-before relation as a simple way to capture it. Next, we present the traditional mechanisms that use this happens-before relation as a basis for a logical time reference, upon which events can be partially ordered. It then presents a brief survey of efficient implementations of these logical clocks mechanisms for the scenario of a well-connected static set of processes where events can occur. Dynamic environments, however, challenge some of the assumptions of the traditional mechanisms, and in most cases preclude their usage. The chapter concludes by presenting some of the most interesting approaches to causality tracking for scenarios of weakly-connected dynamic set of processes.

2.1 System Model

We assume a traditional system model, a distributed system consisting of a set of n asynchronous sequential processes $p_1, \ldots, p_n$. The local states of these processes are disjoint, that is, they do not share a common memory. Processes communicate solely by message-passing over a communication network with an arbitrary topology. Communication is asynchronous, having a finite but unpredictable delay. Also, processes do not share a global clock that they can access instantaneously. A process can execute a computing event spontaneously, and when sending a message, it does not wait for its reception. Events are perceived to be discreet: they are atomic and instantaneous at the level of observation. Sends are unicasts, since messages are received by a single process.

Such a broad model is well suited for describing the general case of a traditional distributed system, where processes execute in a heterogeneous set of hardware, and communicate using an heterogeneous infrastructure, both subject to varying loads in time, affecting the processing speed, and the communication delay. It is unassuming regarding the relevant computing events, which is a useful feature for modelling mechanisms that handle different sets of operations. If needed, it can also be extended in order to support multicast and broadcast operations, as an atomic composition of a sequence of send events. Finally, it can be easily adapted in such a way that it is able to cope with a dynamic set of processes, an inherent feature of the autonomous operation scenario motivating this work.

2.2 The Importance of Causality

A global clock is usually not available due to the inherent limitations of a distributed system. Even if we consider that a common clock is available to all processes in the system—using an external GPS source, for instance—processes could actually perceive different moments in time as the same instant due to unpredictable delays caused by hardware limitations such as interrupt handling, and to unpredictable application loads. Similarly, even if we try to synchronise the physical clocks available to each computer, they can drift from physical time at

different rates due to technological limitations of both their hardware and software.

The absence of global clock makes it harder to reason about the temporal order of events and to collect up-to-date information regarding the state of the entire system. Still, for a number of problems, the hypothetical availability of a global time reference would not help achieving their goals, because it does not help establishing which events are able to influence each other. Since it would order all events with respect to each other, this way of describing executions is not able to capture the situation in which two computation events have no influence on each other, that is, their inherent concurrency. The structure of causality would then have been lost.

Causality tracking in a distributed system is a powerful concept in reasoning and analysing about a computation, and in the design of applications. It is key to solving a wide range of problems in distributed systems, such as in distributed algorithm design [CL85, Mat87], in tracking of dependent events [CM91, HW88, MN91], knowledge about the progress of the system [WB84, AAB04], and as a concurrency measure [CB89, Fid91]. An extensive discussion of causality and its applications can be found in Schwartz and Mattern [SM94].

2.2.1 The Happens-Before Relation

The happens-before relation, introduced by Lamport [Lam78], captures the notion of one event happening in the past of another. This notion translates to the ability of one event to influence others happening in its future, that is, the notion of being their potential cause. The term *potential* relates to the fact that an event happening in the past of another does not necessarily mean that it actually influences it, or is its cause. Proper causality can be established only using semantic information available to the application level, and it is usually more difficult to determine. Potential causality is, however, consistent with it, since it extends the order, possibly only relating more events. Throughout this chapter, unless noticed otherwise, we will refer to potential causality simply as causality.

Consider two events e_1 and e_2 by the same process. Event e_1 can causally influence e_2 if e_1 occurs before e_2, since each process executes sequentially. The

only way for one process to influence another, however, is by sending a message to the other one. That is, an event e_1 occurring at process p_i causally influences an event e_2 of process p_j, if e_1 is the event that sends message m from p_i to p_j, and e_2 is the event in which m is received by p_j. Finally, events can, of course, causally influence each other indirectly through other events.

This happens-before relation can be summarised in a formal description as follows. Given two events e_1 and e_2, e_1 *happens before* e_2, denoted by $e_1 \rightarrow e_2$, if one of the following conditions holds [Lam78]:

- e_1 and e_2 are events by the same process and e_1 occurs before e_2;
- e_1 is the send event of the message m from p_i to p_j, and e_2 is the receive event of m by p_j;
- There is exists an event e such that $e_1 \rightarrow e$ and $e \rightarrow e_2$.

The first condition captures the causality relation between events of the same process. Causality between events of different processes is captured by the second condition. The third condition induces transitivity.

The important property of the happens-before relation is that it characterises potential causality. Moreover, its definition already suggests that mechanisms may be able to track it using local information available to each process and exploiting the communication flow between them.

2.2.2 Logical Clocks

One possible way to capture the happens-before relation is to attach a tag to each event that takes place in the distributed computation. A clock LC_i is available at each process p_i and can be thought of as a function that assigns a value to each event occurring at p_i. The value assigned to an event e is its tag or *timestamp* $TS_e := LC_i(e)$. There is no relation between the values assigned by the clocks and the physical time, hence this approach is usually called *logical clocks*. The logical clocks take monotonically increasing values. In order to capture the happens-before

relation, an irreflexive partial order $<$ on the timestamps must also be provided, in such a way that for every pair of events, e_1 and e_2, if $e_1 \rightarrow e_2$, then $TS_{e_1} < TS_{e_2}$. This is known as the *clock condition* [AW04].

2.3 Classical Causality Tracking Mechanisms

2.3.1 Scalar Clocks

The algorithm used to maintain Lamport's logical clocks [Lam78] can be described as follows. Each processor p_i keeps a local variable SC_i—its logical clock—, which is a non-negative integer counter, initially set to 0. When event e happens at process p_i, it updates its logical clock SC_i as follows:

- If e is an internal computation or send event, it increments SC_i, $SC_i := SC_i + 1$;
- If e is the send of a message m, p_i attaches it a timestamp TS_m set to the current value of SC_i, $TS_m := SC_i$;
- If e is the receive of a message m with a timestamp TS_m, p_i updates $SC_i := max(SC_i, TS_m)$.

The timestamp associated with an event e, TS_e, of process p_i, is the new value SC_i computed during the event. The order on timestamps is the ordinary $<$ relation among integers.

For each process p_i, the value of SC_i is strictly increasing. Therefore, if e_1 and e_2 are events by the same process and e_1 occurs before e_2, then $TC_{e_1} < TC_{e_2}$. Furthermore, the logical timestamps of the event in which a message is received is at least one greater than the logical timestamp of the corresponding message send event. Therefore, if e_1 is the send event of the message m from p_i to p_j, and e_2 is the receive event of m by p_j, then $TS_{e_1} < TS_{e_2}$. These facts, together with the natural order of integers and the transitivity of $<$, clearly imply that if $e_1 \rightarrow e_2$, then $TS_{e_1} < TS_{e_2}$.

2.3.2 Vector Clocks

The converse, however, is not true: it is possible that $TS_{e_1} < TS_{e_2}$ but $e_1 \not\rightarrow e_2$. The problem is that the happens-before relation is a partial order, while the scalar timestamps are totally ordered. Therefore, information about non-causality is lost. Capturing this information requires the logical timestamps to be chosen from a domain that is not totally ordered; such is the case with vectors of integers.

Let's start by defining non-causality more precisely. Two events e_1 and e_2 are *concurrent*, denoted by $e_1 \parallel e_2$, if $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$. Intuitively, processes cannot tell whether e_1 occurs prior to e_2 or vice versa, and in fact it makes no difference which order they occur in.

Mattern [Mat89c], and Fidge [Fid89] independently described a Vector Clock mechanism providing a way to capture causality and non-causality. This logical clock mechanism was probably the one with the most profound practical impact, inspiring several derivative works that cope with specific requirements.

It follows a brief description of the Vector Clock mechanism. Each process p_i keeps a local n-element array VC_i called *vector clock*. Each element is a non-negative integer, initially set to 0. Every time an event e occurs at p_i, it updates VC_i applying the following rules:

- If e is an internal computation or send event, p_i increments the local counter element of VC_i as in $VC_i[i] = VC_i[i] + 1$;
- If e is the send event of a message m, p_i attaches it a timestamp TS_m set to the current value of VC_i, as in $TS_m := VC_i[i]$;
- If e is the receive event of a message m with a timestamp TS_m, p_i updates VC_i with the pair-wise maximum of VC_i and TS_m, as in $VC_i := max(VC_i[k], TS_m[k]), \forall k \in 1, \ldots, n$, and then it increments its local counter as in $VC_i[i] = VC_i[i] + 1$.

The *timestamp* of an event e is the value of VC_i after the rules above were applied. In a sense, for any pair of processes p_i and p_j, the value of $VC_j[i]$ is an

estimate, maintained by p_j, of $VC_i[i]$, according to the information that reached p_j so far. Only p_i can increase the value of the i-th coordinate, and therefore, for every process p_j, in every reachable configuration, $VC_j[i] \leq VC_i[i]$, for all i, $1 \leq i \leq n$.

For scalar timestamps, we had the total ordering of the integers. For vector timestamps, which are vectors of integers, a partial ordering can be defined. Let v_1 and v_2 be two vectors of n integers. Then $v_1 \leq v_2$ if and only if for every i, $1 \leq i \leq n, v_1[i] \leq v_2[i]$; and $v_1 < v_2$ if and only if $v_1 \leq v_2$ and $v_1 \neq v_2$. Vectors v_1 and v_2 are *incomparable* if neither $v_1 \leq v_2$ nor $v_2 \leq v_1$.

Vector timestamps are said to *capture concurrency* if for any pair of events e_1 and e_2 in any execution, $e_1 \parallel e_2$ if and only if VC_{e_1} and VC_{e_2} are incomparable.

Suppose event e_1 occurs at process p_i in an execution and subsequently event e_2 occurs at p_i. Each entry in VC_i is non-decreasing and furthermore, since e_1 occurs before e_2 at p_i, $VC_{e_1}[i] < VC_{e_2}[i]$, for every i. This implies that $VC_{e_1} < VC_{e_2}$.

Now consider in an execution, e_1 the sending of a message m with vector timestamp TS_m by p_i, and e_2, the receipt of this message by p_j. During e_2, p_j updates each entry of its vector to be at least as large as the corresponding entry in T_m, and then p_j increments its own entry by one. Therefore, it is true that $VC_{e_1} < VC_{e_2}$.

These two facts, together with the transitivity of the less than relation for vectors, imply that $e_1 \rightarrow e_2$, then $VC_{e_1} < VC_{e_2}$.

Consider two concurrent events e_1 and e_2 occurring in two distinct processes, p_i and p_j, respectively. Suppose $VC_{e_1}[i]$ is t. Then $VC_{e_2}[i]$ must be less than t, implying that VC_{e_1} is not less or equal than VC_{e_2}, since the only way processor p_j can obtain a value for the i-th entry of its vector that is at least t is through a chain of messages originating at p_i at event e_1 or later. But such a chain would imply that e_1 and e_2 are not concurrent. Similarly, the j-th entry in VC_{e_1} must be less than the j-th entry in VC_{e_2}. Thus, if $VC_{e_1} < VC_{e_2}$, then $e_1 \rightarrow e_2$. This is called the *strong consistency* condition [AW04].

Thus it follows that

$$VC_{e_1} < VC_{e_2} \quad \text{iff} \quad e_1 \rightarrow e_2$$

and that

$$e_1 \parallel e_2 \quad \text{iff} \quad VC_{e_1} \not\leq VC_{e_2} \wedge VC_{e_2} \not\leq VC_{e_1}$$

that is, events are incomparable. Hence, vector clocks capture concurrency.

2.3.3 Version Vectors

Parker et al. [PPR+83] introduced Version Vectors, a causality tracking mechanism sharing an equivalent structure with vector clocks. Its main purpose is, however, the detection of mutual inconsistency between replicas in optimistic replication systems [SS05]. The mechanism is simple and intuitive, and became one of the cornerstones of optimistic data management. Version vectors associate to each replica a vector of integer counters that keeps track of the last update that is known to have been originated in every other replica and in the replica itself. The relevant operations in optimistic replication are typically the update of the replica's state, and the synchronisation of two replicas, the latter being perceived as a synchronous event, converging the state of both replicas. Version vector maintenance is, however, slightly different from vector clocks. Considering a system with replicas $r_1, \ldots, r_n$, version vectors are updated as follows:

- When replica r_i is updated it increments the local entry of its version vector $VV_i[i]$ as in $VV_i[i] := VV_i[i] + 1$;
- When replicas r_i and r_j are synchronised, both version vectors are updated as in $VV_i = VV_j := max(VV_i[k], VV_j[k]), \forall k \in 1, \ldots, n$.

Most of the time, version vectors are used to track causality between coexisting replicas, that is, between replicas forming a consistent cut of the system state [SM94, ABF02a]. In these cases, however, version vectors are overly expressive since, as vector clocks, they record information that enables them to assess causality and concurrency in the past of each possible consistent cut. In the typical

case where past can be discarded, a bounded representation of causality can be achieved using Bounded Version Vectors [AAB04].

Although having similar data structures, Version vectors cannot be based on the set of operations and update rules defined for standard vector clocks. Consider, for a moment, that we try to model the synchronisation of replicas r_i and r_j as the atomic composition of a message sent from r_i to r_j, followed by a message from r_j to r_i. At the end of the synchronisation operation, the vector clocks of both replicas would share the same values for each non-local elements. The elements at position i, however, would have different values since r_i would have incremented its local element once again after receiving the message from r_j. That is, $\forall x \in \{1, \ldots, n\} \setminus \{i\}, VC_i[x] = VC_j[x]$, but $VC_j[i] < VC_i[i]$. In the end, the states of the two replicas would have been perceived as causally related.

2.4 Vector Clocks and the Problem of Scale

Vector clocks are a useful tool in understanding the behaviour of a distributed system. For instance, in optimistic replication [SS05], they are extensively used in replica reconciliation by detecting conflicts [PPR+83, GHM+90, RRP97], and determining the precise set of deltas that must be exchanged in order to progress to a specific state [PST+97, YV01].

Its structure, however, has no bound on its size. More importantly, it grows linearly with the number of processes in the system, and each counter grows logarithmically with the number of events known to each process. The first issue can be a problem for systems with a large number of processes. Although the vector clock mechanism requires no extra communication steps to establish causality between events, messages exchanged between processes are attached with a timestamp of the sender's vector clock. This message overhead can be extremely high for large-scale distributed systems, effectively surpassing the size of the messages themselves. Storage overhead will also be paid for the vector maintained at each process. Moreover, in situations where events are required to be logged by the processes—for instance, preparing state reconciliation in optimistic repli-

cation systems—this overhead can be severe. Although usually not as critical, the computational power will also raise with the size of the vector timestamps, thus playing its own role in the vector clock poor scalability story.

The second issue can be a problem for systems with a high rate of events, and eventually, for any long-running distributed computation. It stems from the fact that processes need to count the number of events that they know to have happened, either locally or at other processes. By the very nature of the problem, being able to compare all events occurring in the system implies that there can be no bound on the size of the counters. Concrete implementations, however, usually resort to fixed-size, *large-enough* counters for the expected duration of the distributed computation, a choice typically rooted in computational performance. Oversized counters and fixed length encoding will, of course, contribute to message and storage overheads.

In the light of these scalability problems, one might be inclined to devise a more compact mechanism that would still be able to characterise the causal precedence relation between events. Charron-Bost, however, proved [CBDGF95] that given a distributed system with n processes, there is always a possible combination of events whose causality can only be captured by vector clocks with n entries. It may be possible to design a different mechanism to determine causality between events, but its structure will still have a size $O(n)$. This result discourages any such attempt.

2.5 Static, Well-Connected Environments

This section presents an overview of relevant work on efficient causal dependency tracking mechanisms for systems with a static set of well-connected processes. In these systems, communication and storage efficiency are usually the main implementation focus. In order to better scale with the number of processes in the system, they explore different trade-offs between message and local storage overheads, and the readiness in which causality can be established. It starts by introducing the differential approach to vector clock implementation. Next, it introduces an

approach to causality tracking providing a solution whose size is independent of the number of processes in the system, at the expense of loosing concurrency information. Later on, Section 2.6 will discuss some of the challenges raised by dynamic environments, and how they are tackled by existing solutions.

2.5.1 The Differential Approach

Singhal and Kshemkalyani introduced in [SK92] a technique for implementing vector clocks that can substantially reduce the communication overhead caused by the timestamps attached to messages. It exploits an observation regarding locality of communication in what is perceived to be a typical distributed computation: at any point in time, only a fraction of the processes in the system are likely to interact frequently. This means that, between successive events occurring at a process, only a small set of its vector clock entries are likely to change. This technique proposes a differential approach to message timestamping that, instead of transmitting the current complete value of the logical time, it conveys only the sender's vector clock entries whose value has changed since it last sent a message to the given particular process. For this communication pattern, sending incremental changes in the timestamp can thus achieve a significant reduction of message overheads, in particular, in systems with large sets of processes.

In this technique, each process p_i maintains three vectors of integers, each of size n: vector clock VC_i, "last updated" vector LU_i, and "last sent" vector LS_i. Logical time is tracked by VC_i, almost as in the original Fidge-Mattern technique [Fid89, Mat89c], the sole difference being that message timestamps only convey a portion of its sender's vector clock. In this technique, instead of a fixed-size vector, timestamps are a set of tuples relating process identities and event counts. The vector clock VC_i is updated as follows:

- When an internal or send event occurs at process p_i, $VC_i[i] := VC_i[i] + 1$;
- When a message m with timestamp $TS_m = \{(id_1, ec_1), \ldots, (id_k, ec_k)\}$ is received by p_i, for each (id_h, ec_h) in TS_m, the vector VC_i is updated as in $VC_i[id_h] := max(VC_i[id_h], ec_h)$, followed by $VC_i[i] := VC_i[i] + 1$.

Vector LU_i tracks for each entry of VC_i the local component of logical time when its value is updated. Similarly, vector LS_i records the local component of logical time when a message is sent to each process in the system. More specifically, LU_i and LS_i are updated according to the following rules:

- When the value of $VC_i[j]$ is updated, $LU_i[j] := VC_i[i]$
- When a message is sent to p_j, $LS_i[j] := VC_i[i]$

At any point in time, the set of position indexes of VC_i whose value has changed since the last message sent to process p_j is defined by $\{k \mid LS_i(j) < LU_i(k)\}$, and it follows that:

- When process p_i sends a message m to process p_j, it attaches it timestamp $TS_m := \{(k, VC_i[k]) \mid LS_i(j) < LU_i(k)\}$

Singhal-Kshemkalyani's technique, however, is only able to replace the original vector clock implementation where FIFO communication channels are available. If communication channels between processes are non-FIFO, message overtaking can occur because reception does not necessarily respects the sending order. In these cases, processes would not be able to correctly reconstruct the sender's timestamp.

We conclude this discussion noting that the Singhal-Kshemkalyani's technique bears some resemblance to previous work published by Mattern [Mat89b]. Both techniques require each process to maintain a vector clock of size n, but in Mattern's work, instead of two extra vectors, it keeps a complete timestamp for each process it sends a message to. This means that regarding storage requirements, Singhal-Kshemkalyani's technique is clearly more efficient, consuming only $O(n)$ space at each process, when compared to Mattern's $O(n^2)$ consumption. Hélary et al describe in [HRMB03] both an improved differential technique that achieves higher efficiency regarding overall message and storage overhead, and an extension to it that is able to cope with non-FIFO communication channels. The set of protocols they introduced is intended to be used by an adaptive timestamping software layer whose purpose would be to select the technique minimising the message overhead for particular distributed system and application.

2.5.2 The Plausible Clock Approach

In order to circumvent the vector clock scalability problem, Torres-Rojas and Ahamad [TRA99] devised a solution that is able to bound the size of a vector clock to a constant value that is less than n. This feature is obtained at the expense of ordering events that may actually occur concurrently in the system. Causality information, however, is not lost: causal related events are always perceived as such by comparison of their timestamps. The authors coined this solution *Plausible Clocks*, since the resulting order does not contradict the causal precedence relation. This order is consistent with the weak clock condition of Lamport scalar clocks: in fact, as we will see below, scalar clocks can actually be perceived as a particular implementation of a plausible clock. As with scalar clocks, plausible clocks do not guarantee the the strong clock condition, also.

Plausible clocks, however, are able to provide much better accuracy than scalar clocks, with much lower rates of wrongly perceiving concurrent events as causal related. The concept of plausible clock actually combines ideas from scalar and vector clocks in order to build clock mechanisms with intermediate strength [Val93]. Their trade-off between scalability and accuracy makes them a practical solution for situations where it is acceptable to deal with an inaccurate but consistent ordering of events with respect to causality in large scale distributed systems.

The plausible clock solution follows a space-folding approach where, instead of exclusively tracking the number of events known from a specific process, each element tracks events occurring in one or more processes. This means that some or all vector elements are effectively shared between processes, and some concurrent events will be ordered and perceived as causally related.

One example of a plausible clock is the *R-Entries Vector* (REV) introduced in [TRA99]. This mechanism can be briefly described as a simple adaptation of the traditional vector clock mechanism, where entries are shared among processes. Consider a system with processes $p_1, \ldots, p_n$, each having a vector clock C_i, whose size is bound to a constant $r < n$. Let f_r be a deterministic function from $\{1, \ldots, n\}$ to $\{1, \ldots, r\}$, mapping each process identity to a specific position in the vector clock. Each clock is maintained as in traditional vector clocks as follows:

- When an internal or send event occurs at process p_i it updates its vector clock to indicate that it has progressed, in such a way that $C_i[f_r(i)] = C_i[f_r(i)]+1$.
- When a process p_i sends a message it piggybacks it with the current value of its vector clock.
- When a process p_i receives a message m from process p_j with a timestamp TS_m, for each x in $\{1, \ldots, r\}$, $C_i[x] := max(C_i[x], TS_m[x])$, and $C_i[f_r(i)] = C_i[f_r(i)] + 1$.

An obvious method for deterministically associating process identities with vector entries is the modulo r mapping, $f_r(i) = (i \mod r) + 1$, but many others are equally viable. All processes p_i such that $f_r(i) = k$ will share the same k-th entry of the vector clock. This sharing will order some but not all concurrent events. In fact, if $r = 1$, then $\forall i : f_r(i) = 1$, and all processes share the single entry of a vector clock whose size is 1. In this case, we actually get the original Lamport's scalar clock.

At the other side of spectrum, if $r = n$ and $\forall i : f_r(i) = i$, we get classic vector clocks, accurately tracking causality. In this case, the vector clock entry i is private to process p_i in the sense that only p_i can entail its increase.

Experimental results show that for a client/server pattern of communication, with up to 3 servers and 100 processes, and for a REV clock of size 2, the number of situations in which $a \rightarrow b$ is falsely inferred from $C(a) < C(b)$, with a and b being actually concurrent, are between 15 and 20 percent, approximately. This error rate is also shown to decrease linearly with the size of the REV clock. For a REV clock size of n the resulting mechanism is the traditional vector clock. In this case, the error rate is zero, since vector clocks accurately capture the causal precedence relation. For a pattern of operation in which processes randomly communicate with each other, and for the same 2-REV clock, the error rate is, however, much higher, reaching a value of nearly 56 percent. This results can be explained by the fact that in the client/server pattern, concurrency is inherently lower than in the random communication pattern, the servers causally relating most of the events occurring in the system. In fact, results also show that the error rate rises with number of servers, due to the increased concurrency.

Another interesting example of a Plausible clock mechanism is also introduced in [TRA99]. The *K-Lamport Clock* (KLA) is an extension of Lamport Clocks, where each site has a vector clock of a constant size $k < n$, keeping a collection of the maximum values of a scalar clock known by itself and by processes that directly or indirectly communicate with it. The KLA clock mechanism has a very different set of update rules and comparison than the REV plausible clock, an a slight lower message overhead. Its main feature is that, for small vector sizes, it is able to achieve lower error rates than the REV mechanism. For the same client/server pattern above, the KLA mechanism is between 7 and 14 percent, approximately, and for the random pattern scenario, it rises to a value of 52 percent. Contrary to the REV clock, however, its error rate drops abruptly and becomes stable from very small vector sizes. Plausible clock mechanisms can also be combined resulting in a new plausible clock. This property can be exploited in order to achieve better accuracy as also shown by experimental results.

2.6 Dynamic, Weakly-Connected Environments

In the previous section, a number of techniques have been briefly described, highlighting some of the trade-offs explored in the design of efficient mechanisms for causality tracking in static, well-connected environments. In these environments, knowledge of the set of processes where events take place is a key factor in achieving efficiency. Process identifiers are assumed to be known beforehand, and they can be deterministically ordered by all processes in the system. Since the set of processes is static, their position in the order is fixed, which enables the use of compact fixed-size structures—such as vectors—to track information regarding each process. Even when variable-size structures are used, they usually explore the knowledge of the process position in the order, avoiding the use of more expensive identifiers. Reliance on the information available in other processes, can also be explored to complement local knowledge. The ability to run a global protocol also enables garbage collection of non-relevant information, if needed.

When systems are dynamic, however, the size and composition of the set of processes in the system will vary over time. In these environments, dependency

tracking mechanisms must accommodate the creation and retirement of processes. In the typical case where there is no a priori knowledge of process identities, in a well-connected environment, processes can resort to the execution of a global algorithm to assign unique identifiers, to fix their position in the global order, enabling processes to adapt the data structures they maintain. As the number of processes increase, however, so does the costs of such procedure, to the point that it may simply become impractical. For large-scale systems, communication, storage and processing overheads preclude the assumption that processes will be able run algorithms involving all processes in the systems. Even if these costs were acceptable, network-partitioning is inherent to these environments, which means that mechanisms must be devised with higher autonomy of operation, such as in the case of process identity management.

The next sections will briefly describe some of the most relevant approaches to causality tracking in weakly-connected, dynamic environments.

2.6.1 Causality Tracking in Roam

The Roam [RRP99] file replication system, was designed to fulfil what was perceived to be three fundamental requirements regarding mobility: the need for direct synchronisation between any two replicas in the system; the support for large numbers of replicas; and the selective control over the files residing on each replica's local storage. In order to achieve these goals, Roam combines elements from the peer architecture and the client-server model into what was called the Ward Model [RPR96]. In this model, replicas are clustered into groups called *wards* (wide area replication domains) that try to capture the notion of typical communication partners. Replicas can be part of any number of wards, and move between them without the need for global coordination. Within wards, communication is cheap when compared to the communication with the outside world, and all replicas are peers that can synchronise directly. Consistency between wards is maintained by an automatically elected master from the available peer replicas, effectively participating in two or more wards. The master has, actually, a very lightweight role: it is solely responsible for propagating updates regarding

the different sets of files being replicated by its peers. Intra-ward communication is assumed to be more expensive but less frequent than communication between peers. This model helps ROAM to better fulfil the intended mobility requirements, specially when compared to client-server replication approaches, such as CODA [SKK+90] and LITTLE WORK [HHRB92]. At the same time, the Ward Model helps to achieve better scalability than previous approaches to peer-based replication, such as BAYOU [PSTT96], FICUS [GHM+90] and RUMOR [GRR+98].

Update tracking in ROAM is based on the Version Vector mechanism, but uses an innovative technique [RRP97] that can help minimise their scalability problems. The devised technique exploits two key observations. First, updates usually occur in a few isolated *hot-spots* at any point in time. While the hot-spots may change over time, it is rare to see a replicated object being frequently updated by everyone, a notion well supported by Wang's work [WRB99] on productivity environment data taken at LOCUS COMPUTING [KPR94]. Experience from Ficus and Rumor also shows that some replicas never generate updates. The second observation is that once an element has the same value in all replicas, it is no longer relevant for its comparison, that is, for establishing causality or concurrency between replicas. These observations lead to the idea of dynamically expanding the version vector once updates are generated by each replica, and periodically compressing it by extraction of all equal value elements.

For this end, instead of pre-allocating vector elements for each replica, as a replica generates its first update, the vector is expanded by adding the required new element. Zero-value elements can be trivially removed from the vector since they are insignificant for causality tracking. Once all replicas have the same value for a certain element, it can also be removed from each vector with no loss of distinguishable comparison information. This technique, however, preserves the ability for a replica to generate new updates later on, an as such, this element can then be added to the vector as it becomes relevant once again for causality tracking. Updates generated by *cold* replicas are infrequent by definition. This means that their corresponding element's value will quickly stabilise and propagate to the other replicas, and as such it will be aptly removed from the vector the next time the compression process is executed. Since at any point in time the version vector

will tend to keep only the elements required to establish causality or concurrency, and that hot-spot replicas are assumed to be just a small fraction of the total, this technique offers a potentially significant reduction of the vector size when compared with the original approach.

This expansion and compression technique, however, requires a number of modifications to the original version vector structure. First of all, a replica's position in the vector can no longer be identified using a pre-defined mechanism: a replica can have no entry in the vector timestamp, its position can change over time as other entries are added or removed from the vector. This can be either either because a cold replica became active, a new replica joined the system and generated its first update, or a compression was performed. This means that a vector can no longer be used to represent the map from replica identity to its counter value. Instead, this technique uses an associative array indexed by replica identifier.

Extra information must also be recorded in order to periodically run the compression process in such a way that: the system does not block and it is safe to generate updates during its execution; it can operate on multiple elements; and finally, that it can be used to remove the minimal (common) value known to all replicas, while preserving the information required to establish that certain replicas have seen more updates than others. The algorithm used in this technique can be briefly described as follows. For each element, the algorithm achieves a consensus on the value to be subtracted from the existing element. After the consensus is established, each replica subtracts this common value from the respective element, and when they become zero they are removed from the vector. Correctness is guaranteed by tagging each version vector with the number of times compression has occurred. This extra counter prevents comparison of elements when one has already been compressed and the other has not. A spare counter is associated with each element being compressed, allowing consensus to be reached independently of new updates.

2.6.2 Causality Tracking in Bayou

Bayou [PSTT96] is an optimistic replication [SS05] database system designed primarily for mobile computing environments. Among its goals, of particular interest to this survey is Bayou's approach to causality tracking, while coping with its scalability and high availability requirements. Bayou is particularly relevant because these requirements are intended to be met in an environment consisting of a potentially large and dynamic set of replicas, and where connectivity is intermittent. In order to accomplish this task, Bayou resorts to a weakly consistent replication model, and an anti-entropy propagation protocol, allowing replicas to diverge while moving towards eventually consistent states. This is achieved by the total propagation of writes, their consistent ordering and deterministic execution on each replica [TTP+95].

Bayou proposes an operational environment where clients are able to read and write any replica in the system. Writes from clients on a given replica propagate to other replicas strictly using pair-wise communication. New replicas can be created from any existing one, with no need for a centralised or global consensus protocol. Knowledge of newly created replicas relies on the same anti-entropy propagation protocol used for writes. This operational pattern actually reveals a key design decision: operations must not involve more that two entities. This also means that the cost of these operations is unaffected by the addition of new replicas in the system. In Bayou, entities operate with high autonomy, which translates into better scalability and availability. Inevitably, autonomy also means further potential for divergence between replicas. The following paragraphs will briefly describe Bayou's weakly consistent model, highlighting how it tracks causal dependency between the set of write operations known to each replica, and how it copes with a varying set of replicas.

Each replica accepts write operations from clients, logging and applying them to the database in the same global order. Each write is assigned a *write-stamp* value, a three tuple (*commit-stamp*, *accept-stamp*, *replica-id*). Its initial value is, respectively, the infinity value, the value of a monotonically increasing local counter, and finally, the replica's identifier. This write is deemed as tentative, and its

position in the global order can vary as the replica learns of other write operations from other replicas. Later on, it eventually receives a final commit-stamp, an also monotonically increased counter value, this time obtained from a *primary replica*. When a write becomes committed, it finalises its position in the global order of writes. Committed writes are totally ordered according to the timestamps assigned by the primary replica, and precede all tentative updates in the global order. This strategy also means that, in order to guarantee eventual consistency, replicas must be able to rollback the effects of previously executed writes and redo them according to the evolving global serialisation order. Bayou allows clients to observe this reordering. In fact, unlike other optimistic replication systems such as Ficus and Coda, it does not try to provide transparent replication. Clients must accommodate this inherent reordering of writes at each replica. In Bayou, conflict detection and resolution is perceived to be a semantic issue, and is left to be handled at the application level.

Update propagation honours a *prefix property* that states that if a replica R_1 holds a write W_x accepted from a client by a replica R_2, then R_1 also received all writes accepted by R_2 prior to W_x. This means that all writes accepted by R_2 whose accept-stamp is smaller than the one of W_x must also be known to R_1. This property enables a compact representation of the replica state since it suffices to keep track of the largest accept-stamp assigned by each replica. In Bayou, replicas record this information in a local *Write Vector*, a version vector-like structure, mapping replica identifiers to their largest known accept-stamps. Their role is to capture the causal precedence relation between writes among the replicas in the system.

The main difference regarding traditional version vectors, however, relates to Bayou's support for a varying set of replicas in the system. In Bayou, the write vector of a given replica grows and shrinks according to the set of active replicas known to it. Knowledge of both the creation and the retirement of replicas is propagated according to the same anti-entropy epidemic protocol that is used for write propagation.

A replica R_i creates itself by sending a creation write to any replica R_l that may be available to it. As with any other write, replica R_l assigns it a local write

stamp of $(\infty, AS_l, R_l >$, AS_l being the current value of its accept-stamp, and records it in the write log. The identifier of the new replica R_i becomes the tuple $(AS_l,\ R_l)$. Since accept-stamps are monotonically incremented at each replica, these recursively created identifiers are globally unique.

These identifiers have also the interesting property that enables them to be used to track the replica's lineage, and in particular the moment in time each of its ancestors where created. Once R_l knows about the newly created replica, it adds its identity to its write vector, and propagates this knowledge, as it does for any other write from the set of active replicas it knows. Retirement of replicas follows the same procedure. A replica R_i sends a *retirement write* to replica R_l. From then on, it stops accepting write operations from clients. It is assumed that it keeps running long enough in order to propagate its retirement to at least one other replica. When a replica receives a retirement write accepted from other replica, it removes its entry from its write vector.

Committed writes, however, are allowed to be removed from replicas write logs in order to save storage resources. This means that creation and retirement writes may never reach some of the replicas in the system. When two replicas exchange information during the anti-entropy protocol, some write vector entries may be unknown to one another. A replica identifier may be absent from a write vector in one of the two situations: either a replica never heard about the missing replica, or it learned about its creation and subsequent retirement. Fortunately, since the identifiers can be used to track the replica's lineage, these situations can be disambiguated. Consider replicas R_1 and R_2 exchanging their write vectors during anti-entropy. If R_1 has an entry regarding a replica $R_i = (AS_j, R_j)$ unknown to R_2, and R_j is known to R_2:

- If $R_2.WriteVector(R_j) \geq AS_j$, then R_2 has seen R_i's creation write, and R_1 can remove its entry from its write vector;
- If $R_2.WriteVector(R_j) < AS_j$, then R_2 has not seen R_i's creation write, and thus it can not have seen its retirement write either; in this case, R_2 adds R_i entry to its version vector.

Knowledge of R_j by R_2 is, however, non essential. Since identifiers recursively

identify the moment in time a replica is created from its ancestor, the same rules can be applied to an expanded write vector calculated from its current entries.

In spite of its elegant solution, Bayou's identity management and causality tracking method has its limitations and drawbacks. When the set of replicas is large, identifiers can be a long sequence of counters, which will translate into high storage costs in all replicas that know about it, particularly in their write logs. In order to minimise this problem Bayou resorts to the ability of pruning entries from write logs and write vectors, when writes are committed.

Bayou's commit protocol resorts to the use of a primary replica, which partially defeats its goal of autonomy of operation. This is particularly true if this primary fails and its retirement is not propagated. To our knowledge, no hint has been given on how a new primary gets chosen upon failure, but one can speculate that an epidemic protocol would also be used.

Failure of a primary, will certainly have a severe impact in the system, delaying its evolution into eventual consistent states, and increasing the storage resources used by write logs. Another important drawback will be its operation under churn. In this case, identifiers will be created from existing ones, and quickly retired from the system. In this case, the size of identifiers could grow rapidly, particularly if churn occurs over lineages, in which case size grows linearly.

2.6.3 Tree Clocks

Another interesting but limited approach to causality tracking in environments with a dynamic set of processes is the work of Landes on Tree Clocks [Lan07]. It is designed with the goal of scaling efficiently with the creation and the retirement of processes, under a nested model similar to the one described by Fidge in [Fid91]. In this model, the computation starts with a single process, from which new processes can be recursively forked, effectively forming a tree. A process retires by joining with its parent. It is assumed that parents outlive their children, and the computation eventually terminates with the original process that first started it. This creation and retirement model resembles the fork and wait pattern used in Unix process management, although in this case processes can terminate before

their children.

In order to track causality, tree clocks count the number of events of a single process in the node of a tree of counters. Upon creation, counter nodes have an initial value of zero. When the process forks, it increments its counter, and two new nodes are appended as children of its node. Each of the resulting processes inherits the new tree, and immediately (and exclusively) increments one of the child counters. When a process terminates by joining with its father, the father increments its counter and discards the node that was being updated by the terminated child. Discarding this node can be done since no more events can be recorded by the terminated process, and all processes that may have been forked from the terminated process have also been terminated. When a counter node is a leaf with no sibling, it is also discarded, and its ancestor node counter is incremented. This leaf-without-sibling rule is applied recursively in order to simplify the tree.

Events are compared using a breath first search for a pair of nodes in the same position of the two trees holding different counter values. If one of the values is zero, then there is no causal precedence between them, which means they are concurrent. A smaller value implies causal precedence to the other event. If there is no match for a particular node, then it is not taken into account. The first match with different values ends the comparison algorithm.

One may note that one of the counter siblings has always a zero value. This happens because when a process forks, each process will exclusively update one of the new counters, the other remaining with its initial zero value. Tree clocks, however, can be used with message passing. In this case, the value of counters in each tree node can propagate to every coexisting process, that is, it is no longer restricted to the process nesting pattern. On message passing, clock trees are updated as follows. When a message is sent, the sender process increments its counter node; the message is timestamped with the current value of the sender's tree clock. When a message is received, as with any other event, the receiving process starts by incrementing its node counter; then, it updates each of the counters in its tree, with the maximum value between the corresponding nodes in its tree clock and the message timestamp; it also appends missing nodes to its tree clock, as long as they

are not direct descendants of its own counter node. In order to support message passing, the join operation must be slighted modified by not allowing nodes to be appended to the father's clock tree, and by setting the nodes present in both trees to the respective maximum.

Although tree clocks try to provide a natural and a scalable solution to the dynamic creation and termination of processes, its reliance on process nesting seems to be an important limitation to its usage in most scenarios. The assumption that each process outlives its descendants seems unreasonable for the general case, and may lead to processes being artificially kept alive. This will inevitably translate into structural growth that not only propagates to descendants, but also to other processes by message passing. Enforcing processes to terminate by joining with their direct ancestors actually preclude its usage where network partition is frequent, and in particular, where processes must be able to operate autonomously.

2.6.4 Hash Histories

Kang et al introduce in [KWK03] the Hash History approach to causality tracking in the context of optimistic replication. This approach is interesting because it is independent of the number of replicas in the system, while its overhead increases with the number of updates.

In hash histories, causality is encoded as a directed graph of version hashes over the evolving state of each replica's data. In a sense, this can be perceived as a pragmatical adaptation of the intuitive causal history dependency tracking mechanism [SM94]. Causal histories encode causal dependencies as the set of event identifiers that happened before each particular event in the system (including itself), and order them according to set inclusion. They are, however, mainly a theoretical mechanism: as its size grows indefinitely with each event, message and storage overheads would become impracticable. Two differences can be observed regarding causal histories, though. First, hash histories are used to track dependencies among replica states, while causal histories would have been typically used to track dependencies among update events. Second, if causal histories were used, each update event would require a globally unique identifier in order to add it

to the happened-before set. In a network-partitioned environment, this globally unique identifier could be based on a recursive creation technique, as the one used in Bayou [PSTT96]. By contrast, hash histories obtain this update identifier from a calculated hash value of the replica's state. This means that in hash histories, update identifiers and causal dependencies are only probabilistically correct.

Version hashes are also exploited as a way to minimise conflict detection when two operations produce the same output, that is a *coincidental equality*. Contrary to what would have happened if causal histories were used—or traditional version vectors, for that matter—updates are perceived as non-conflicting.

An update event in a replica can, however, generate an hash value that already exists in its hash history. In order to distinguish the new and old versions of a replica, hash histories attach an *epoch* counter to each particular hash value. Still, the same hash value and epoch pair can be generated concurrently by different replicas with actual different states. In this cases, the hash history approach stipulates that they are the same version, that is, it incorrectly states that they have the same exact state, and that it can be used as a basis for future reconciliation.

Replica dominance, that is, establishing if a replica version depends on another, is achieved by set inclusion over the hash histories. Reconciliation starts by establishing the most recent common hash value in the hash histories, and exchanges the missing updates from that point onwards. If there is no such common hash value, reconciliation must be achieved using the current, concurrent states of both replicas.

Since hash histories grow with the number of updates, it is crucial to truncate them in order to minimise message and local storage overheads. To achieve this goal, hash values are pruned using a simple ageing method, based on roughly synchronised clocks. For example, hash values with more than 30 days could be discarded from the hash histories.

If the risk of having only probabilistic guarantees is acceptable, the hash history approach seems to be an efficient causality tracking mechanisms for settings with a large and dynamic set of replicas and a low rate of updates.

Chapter 3

Panasync: Dependency Tracking Among File Copies

File copying is frequently used to implement *ad hoc* management of file replicas, backups and versions. Such tasks can be assisted by appropriate applications, at the expense of introducing some restrictions to the usage patterns. In particular, this is the case of interactions involving disconnected machines and transportable media. PANASYNC tries to support these actions by introducing a set of commands for file copying and re-integration that complement the file-system commands and provide support for dependency analysis among time-stamp assisted files.

3.1 Introduction

User interaction with the file system is supported by command line or by graphical user interfaces, both alternatives providing standard operations such as file and directory creation, renaming, copying and removal. In their normal activity users can resort to a given operation in order to achieve different purposes. For instance, the *copy* operation can be used to create a backup of a file, to branch a project or even to substitute a file with the contents of another. Due to the simplicity of the basic operations available to the user, the underlying purpose cannot be perceived by the system. The system treats the operations indistinguishably, thus having no

provision to assist the user along its tasks. Consequently the user is on its own, regarding, for instance, file version management.

The common solution to this problem is to adopt version-control environments offering a special set of tools and often embedding their own filing structures in the underlying file system. However, the adoption of a special environment for versioning control is usually a matter of complexity assessment, and users tend to avoid it when they want to manage what they perceive as a simple file duplication or versioning task. Few users resort to versioning support tools when taking a draft document copy to eventually work on it in a weekend.

Additionally, the use of existing versioning and replication environments calls for a centralised or at least pre-set distributed configuration that is frequently inadequate to the user mobility needs. Missing features encompass the uncoordinated creation of new replicas and isolated forking of new versions. Together these restrictions can lead users back to basic file system operations, or at least make them think twice before switching into a coordinated environment.

In this article we discuss the design of a set of tools that provide autonomous file copying and versioning. These constructs can be implemented by a set of commands that manage additional time-stamping data over an off-the-shelf file system or by the design of a file system extension that manages and hides the time-stamp data. These tools can be used as a complement to the standard file system operations.

3.2 Related Work

Replicated file systems such as CODA [KS91], FICUS [GHM+90] and RUMOR [RPG+96], are bound to rely on some notion of replication volumes (typically subtrees in a given machine). These systems can be related to version control systems if interpreted as providing version control over a well defined number of branches, when allowing optimistic replica evolution on each volume.

In these systems, the use of vector time-stamps for conflict detection requires either an indexing of the replication volumes or a way to univocally identify each

volume. That assumption enables the design of file time-stamps as mappings from volume *id* (which can be a name or vector index) into an update counter [PPR+83, RRP97]. A consequence of this design is that a file cannot be replicated in the same volume, and in particular in the same directory.

Another problem is the transitive replication that can occur, for instance, when using transportable media. In this case, the availability of one replica is not sufficient for the autonomous creation of the new replication volume that would host a subsequent replica. All these patterns of usage, when actually needed, lead the users back to the use of uncoordinated copy.

The solution to the autonomous identity creation problem relies on a recursive construction of the new ids, in the presence of a single file replica. The BAYOU system [PST+97] uses a similar technique for volume identity creation. PANASYNC will use a recursive technique that is based on previous work on autonomous causality [BM99] and autonomous file time-stamping [BA99] in order to provide single file replication and versioning.

Simple but well deployed forms of replication systems, such as Microsoft Windows Briefcase and the new off-line files and folders of Windows 2000, target optimistic replication between a mobile unit (or an instance of transportable media) and one host. This restricted form of replication fits the general case of replicated file systems.

Traditional versioning systems, such as CVS[1] and PRCS[2], are targeted to manage short derivations from a central branch of development. In this sense they support an arbitrary number of concurrent evolutions but do not treat them as first class elements, and keep a centralised control on the versioning information. A recent trend in versioning systems, as depicted in BITKEEPER[3] design, aims to support parallel lines of development that share code improvements with a more robust 'diffing' technique. This approach differs from the PANASYNC scenario as it targets the management of sets of files (repositories), in contrast with a file based approach, and focuses on 'diff' portability and not on causality based domina-

[1] `http://www.sourcegear.com/CVS/`
[2] `http://www.xcf.berkeley.edu/~jmacd/prcs.html`
[3] `http://www.bitmover.com/bitkeeper/`

create(body,target) `echo "`*body*`" >` *target*
copy(base,target) `cp` *base target*
move(base,target) `mv` *base target*

Table 3.1: Classic file constructs.

tion analysis between parallel evolutions. Nevertheless this trend appears to share common motivation with our approach whilst at a different level.

3.3 Revisiting Copy Constructs

PANASYNC intends to add file based replication and versioning constructs in a way that complements the usual file manipulation constructs. Since some of their functionality will be complemented, it is important to review the possible applications of existing operations. Considering the arbitrary syntax and Unix mapping shown in Table 3.1, we discuss how some user tasks expressing replication or versioning are typically performed.

If the intent is to create a new file unrelated to the other files, the operation could be *create(content,new-file-name)*. However it is sometimes useful to start the new file with the content of another file (for instance when starting a new LaTeX document or a CGI script), which would lead to *copy(file-name,new-file-name)*.

When the intent is to keep a backup copy of a given file, the operation is something like *copy(file-name,file-name.orig)* or slightly different, *copy(file-name,file-name.v01)*, if several backup versions are to be created. If, latter on, the user wants to discard changes he issues something like *copy(file-name.orig,file-name)*. Alternatively if the user needs backup versions for a set of files, he might make use of a new sub-directory that gathers a given version of the files, and then use something like *copy(*,dir-name.v01)*, which keeps the original file names but places them in a different name space.

Finally, when the intent is to replicate a file or set of files the practice is to keep the names and copy them into a different name space (disk and directory). Later on, replica identification and the possibility of replica re-integration must be evaluated by the user and lead to the appropriate *copy* or *move* operations.

In all these tasks the copy operation is heavily used and its different intents are only vaguely captured by the choice of names and name spaces that the user conducts. There is no way of providing system support to these operations and the users are on their own to make either correct decisions or mistakes.

Its is also clear that the directory structure is used for several purposes, division of name space, classification, and identification of different physical storage devices. It can be the case that files like `/floppy/panasync.tex` and `/home/cbm/psart.tex` are versions of the same entity and `/src/q/readme.txt` and `/tmp/qinst/readme.txt` are totally unrelated.

3.4 PANASYNC Operations

In order to assist file replication and versioning tasks, we propose a set of commands that track dependencies between versions of files. A file system with PANASYNC extensions, or user level commands, manages ordinary as well as panasync-enabled files, the latter having an extra time-stamp attribute. For convenience of discourse we designate the panasync-enabled files *pfiles* and the others *ofiles* (ordinary files); we use the term *file* to refer to either class. The traditional commands apply to both file types, but pfiles can also be manipulated by the PANASYNC operations as shown in Table 3.2 (where we also present possible Unix mappings).

The *new* operation is used to create a pfile. Its name *target* is mandatory and an optional file name for initialisation is allowed by indicating a *base* file. Usage of this operation means that new a lineage of files is being created and that the target pfile is not comparable with other lineages. Ordinary files are all non comparable.

When a backup, versioning or replication action is needed, users can resort to the *duplicate* operation. This operation creates a *target* pfile from the *base* pfile and ensures that both share the same lineage. After duplication both pfiles are

new([base],target) `pananew` *base target*
duplicate(base,target) `panadup` *base target*
join(base,target) `panajoin` *base target*

Table 3.2: Some PANASYNC constructs.

equivalent, share the same contents and should be regarded as siblings. In fact, although *base* was the starting point they do not hold a parent/child relation.

An immediately subsequent *join* operation with one of these pfiles as *base* and the other as *target* would remove *base* since the system detects that they share unchanged content, as well as positions that can be determined to be equivalent in the version lineage.

By consulting the pfiles time-stamps the *join* operation is able to relate any pair of files, verify if one of the following conditions holds, and advise appropriate action:

- **Condition**: *base* and *target* are in distinct lineages or one or both of them are ofiles.
  Action: Abort the *join* and do nothing.

- **Condition**: *base* dominates *target* or *target* dominates *base*.
  Action: The normal action is to remove *base* and place in *target* the content that dominates (either from *base* or *target*), but an option can be provided to choose the dominated content.

- **Condition**: Neither *base* nor *target* dominates, which means that they hold concurrent updates.
  Action: Do nothing or prompt the user for a reconciliation file, in which case both *base* and *target* are removed and the new contents are stored in the position *target*. This new file dominates all files that is ancestors would dominate.

This description shows that names are not important in these operations since pfiles have enough information to distinguish file instances as well as to compare them. As a consequence of this, the choice of pfile names need only address name clash avoidance in the directory system that stores them.

In fact it is possible to design an option that applies a *join* operation recursively to two whole subtrees. This would select all pfiles from the subtrees, produce two flat lists of files and compare those in the same lineage, removing, for instance, the dominated files from the first subtree.

Another useful construct, although not a basic one, is a `panasync` *base1 base2* command that produces a *join* of the two files in a temporary pfile and immediately duplicates it again into *base1* and *base2*. The overall effect is to keep two copies with synchronised contents.

Renaming pfiles can be done as usual with the original *move* command, as long as time-stamp association to file name can be tracked. Depending on the system, this need can lead to a simple patch to the native *move* command or two the introduction of a `panamv` construct. The use of the native *copy* with a pfile as *base* produces a *target* ofile with unrelated lineage, which is a useful functionality.

3.5 Synopsis of Time-Stamping

The complex pattern of version and lineage control can only be achieved with a sound time-stamping technique that supports autonomous creation of a partial order among file replicas and the identification of lineages. A presentation of the causality model and time-stamping technique is beyond the scope of this chapter. Some insight on the technique can be found in [BM99, BA99]. Here we will only address some significant points that characterise this time-stamp model.

Vector time-stamps, as originally shown in [PPR+83], allow the tagging of identical replicas with identical time-stamps. This is possible due to the fact that the identity of the replication volumes and the information of the hosting volume for each replica can complement the information stored in the time-stamp. On the contrary, if we wish to have autonomous time-stamps all the relevant information

must be stored in each replica time-stamp. This leads to the existence of distinct time-stamps that identify equivalent replicas. The partial order algorithm must detect that simple replica duplication does not make them different but only raises the possibility of separate modifications. Such replicas cease to be equivalent once they suffer changes.

Unlike vector time-stamps, this scheme does not impose structural limits on the number of replicas, since replica identity is recursively constructed with the information that is locally available.

3.6 Example Scenarios

The example in Figure 3.1 (with first scenario in Figure 3.2) shows a hypothetical use of PANASYNC commands under the Unix environment. In the setup phase a new lineage is created together with the pfile `pana.bib` and its contents are initialised with `mybibs.bib` content. We recall that there is no ordering relation between these two files.

Afterwards the `pana.bib` file is duplicated to a directory mapping a floppy device and its contents are changed with the concatenation of `entry1.bib`. Finally this file is duplicated into `/zip/p.bib`. We can expect that `/zip/p.bib` and `/floppy/pana.bib` are equivalent, and that both dominate the local `pana.bib` content.

For simplicity all examples have been illustrated in a single machine. It must be kept present that all operation steps are possible on any arbitrary machine that accesses the used persistent store. This use of floppy and zip names emphasises this possibility since they designate transportable persistent media.

3.6.1 First Scenario

Now we change the name of the floppy resident file into `/floppy/panasync.bib`. In fact we can *move* this file to any place or system since its identity does not depend on its name. Next we add some content to `p.bib` and try to *join* it with

the local `pana.bib`.

This *join* is straightforward since one of the files dominates the other. As usual, the two file supplied as argument to `panajoin` are checked for their relative order and the *join* outcome is written to the second file argument. This is the case even if the second file is the dominated one.

After the last `panajoin` invocation the three replicas from the start of this first scenario have been collapsed into a single replica at `/floppy/panasync.bib`. Since there were no concurrent changes the convergence was trivially accomplished. A simple way to check for the presence of concurrent changes, in Figure 3.2 as well an in the second scenario figure, it to track the bullets (•) that indicate changes. This can be done by following the arrows, from the replicas, in the reverse direction and check if both have changes that the other has not seen.

3.6.2 Second Scenario

In this second scenario, with its evolution outline in Figure 3.2, we make sure that some concurrency of changes occur, by adding content to both `/zip/p.bib` and `/floppy/pana.bib`. Consequently the *join* tentative over this two files fails and issues a warning identifying the occurrence of concurrency and asking for the provision of a content that re-conciliates the files.

The user is free to choose the content that is to be provided. Here he uses the `sdiff` tool to select the merged content from the two concurrent files and supplies it in the next `panajoin` invocation.

The last `panajoin` invocation illustrates that when merges of concurrent evolution occur, the order of the new pfile is such that it dominates all the pfiles that were previously dominated by either of the merged pfiles. This factor empowers the user decisions when supplying a merge and helps future replica convergence, thus constituting a very powerful property that is particular to this system.

3.7 Design Issues

The basic PANASYNC implementation will be built over a set of portable command-line tools. The purpose of these tools is not only to test the usefulness of this dependency tracking system, but also to ease its integration into existing file managers. A second phase will encompass the exploration of an adaptation technique for native file systems, eventually with the use of a reflection mechanism.

In PANASYNC, file naming is a user convenience although the system does not rely on it to track dependencies between pfiles. Evidences from observation of typical patterns of usage suggest that PANASYNC users should be able to change pfile names at will and the system must still be able to ensure correct dependency tracking. To achieve this purpose, PANASYNC will rely on a mapping from a special pfile identifier into its name. This will enable the system to assess if two pfiles belong to the same lineage independently of their current names. The identifier will be given to the pfile upon creation and associated to the specified name. Each time a *move* is issued the pfile identifier mapping will be updated. For practical purposes this identifier can be generated from traditional techniques based on existing hardware settings (e.g. ethernet address), file creation time and a random value.

To achieve its purposes, PANASYNC needs to store this extended information about pfiles. In fact, not only it will need to record the current name of each pfile, but also its time-stamp and an MD5 digest to actually track its dependencies.

Another issue in the design of PANASYNC is the transparent detection of modification of pfiles' content. This objective cannot be reliably achieved by evaluating the creation and modification time-stamps provided by most of the traditional file systems. Instead, PANASYNC calculates a MD5 digest for each pfile upon its creation, and also stores this information on the mappings discussed above. Each time `panadup` and `panajoin` are issued the MD5 is recalculated, enabling the detection of modifications on the pfiles with the setting of a dirty attribute that is used in the time-stamp construction algorithm.

To ensure portability PANASYNC will provide an external representation of pfiles' attributes. This will enable transferring pfiles through non-supported sys-

tems, such as email, for instance.

3.8 Conclusions and Future Work

We have presented the motivation behind the conception of PANASYNC and shown usage scenarios. The system aims to support common tasks of file replication and versioning, which could be done either manually, without system support, or under control environments that are focused toward coarser grain scenarios. We believe that the addressed patterns of fine grain file copying are bound to increase with ongoing trends of increased user mobility and information sharing among mobile and fixed units. The PANASYNC approach does not intend to substitute the functionality of versioning systems or replicated file systems, but rather act as orthogonal support for a particular and common class of use cases.

Apart from the practical design issues, the central point that enables the conception of a system with these characteristics is the underlying time-stamping scheme. Presently, we have reached a time-stamp design that allows identifier simplification upon joins. This design format allows us to start the construction of the first command prototypes.

Having designed the time-stamping mechanism, the next step will be the study of time-stamp size impact on the system under an average work pattern. Although not comparable with the small size of standard time attributes we are confident that the extended control possibilities will make the use of pfiles worth in a significant set scenarios.

```
.... Setup .....

$ pananew mybibs.bib pana.bib
$ panadup pana.bib /floppy/pana.bib
$ cat entry1.bib >> /floppy/pana.bib
$ panadup /floppy/pana.bib /zip/p.bib

.... 1st Scenario ....

$ mv /floppy/pana.bib /floppy/panasync.bib
$ cat DSM.bib >> /zip/p.bib
$ panajoin pana.bib /zip/p.bib
Info: /zip/p.bib content
      dominates pana.bib
$ panajoin /zip/p.bib /floppy/panasync.bib
Info: /zip/p.bib content
      dominates /floppy/panasync.bib

.... 2nd Scenario ....

$ cat DSM.bib >> /zip/p.bib
$ cat OS.bib >> /floppy/pana.bib
$ panajoin /floppy/pana.bib /zip/p.bib
Warning: Files are concurrent
         use -s to specify substitute
$ sdiff /floppy/pana.bib /zip/p.bib -o merge.bib
$ panajoin /floppy/pana.bib /zip/p.bib -s merge.bib
$ panajoin /zip/p.bib pana.bib
Info: /zip/p.bib content dominates pana.bib
```

Figure 3.1: Example runs with PANASYNC tools.

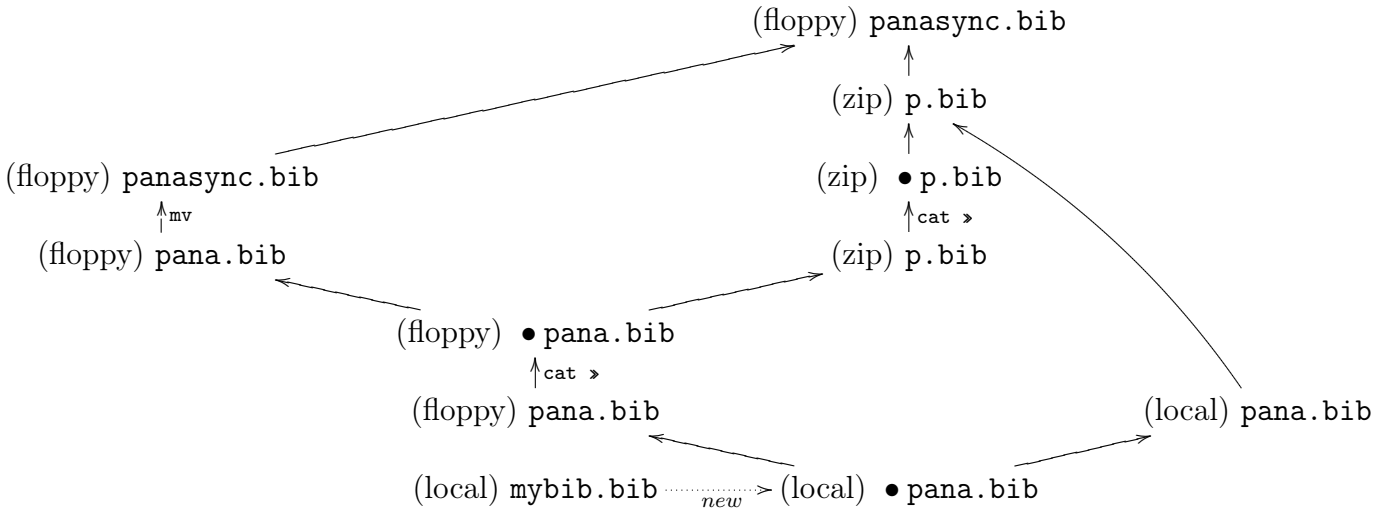


Figure 3.2: First Scenario. Here a single branch dominates the other branches. The `mv` action that renames one of the *pfiles* does not change its identity and time-stamp.

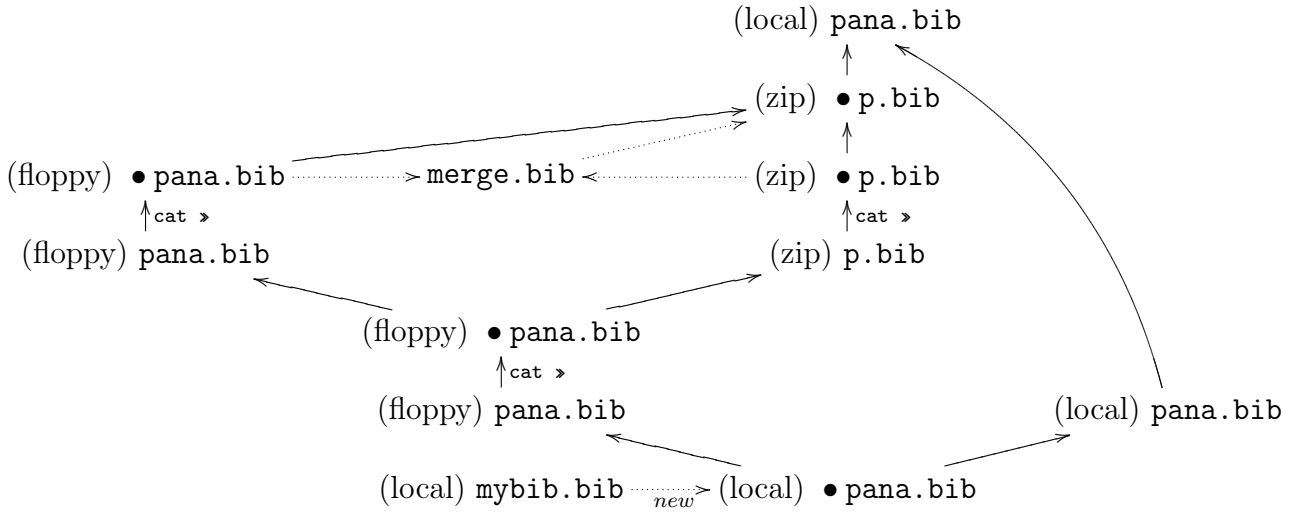


Figure 3.3: Second Scenario. Two parallel branches suffer concurrent changes and are re-conciliated with a merge content. The resulting *pfile* inherits existing domination relations and supersedes an early branch.

Chapter 4

Version Stamps: Decentralised Version Vectors

Version vectors and their variants play a central role in update tracking in optimistic distributed systems. Existing mechanisms for a variable number of participants use a mapping from identities to integers, and rely on some form of global configuration or distributed naming protocol to assign unique identifiers to each participant. These approaches are incompatible with replica creation under arbitrary partitions, a typical mode of operation in mobile or poorly connected environments. We present an update tracking mechanism that overcomes this limitation; it departs from the traditional mapping and avoids the use of integer counters, while providing all the functionality of version vectors in what concerns version tracking.

4.1 Introduction

Mobile computing has evolved in the previous decade into what is now a common mode of operation for a significant share of distributed systems. This mobile context helped to promote optimistic strategies and, with them, the need for version vectors in update tracking. Nevertheless, the same mobile context also brings to surface some of the limitations of version vectors, in particular concerning the iden-

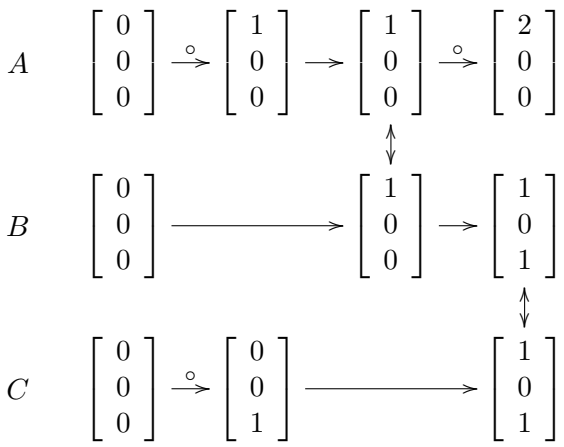


Figure 4.1: Use of version vectors to track updates among three replicas.

tification of participating entities in the computation in such potentially dynamic environments.

The concept of version vector [PPR+83] is connected to the twin concept of vector clock [Fid89, Mat89c], and both are rooted on causality in distributed systems [Lam78]. These concepts share an equivalent structure that consists in a mapping from process/replica identifiers to integer counters, $I \hookrightarrow \mathbb{N}$. In practice, version vectors and vector clocks are more often represented as a fixed sequence of integer counters, $\{1, 2, \ldots, k\} \rightarrow \mathbb{N}$, which is a reasonable choice as long as the number of entities is known in advance. Figure 4.1 shows an execution in a replicated system where fixed size version vectors are used to track updates to each of the three replicas in the system. The direction of evolution is represented by the arrows, with dot annotated arrows, $\overset{\circ}{\rightarrow}$, depicting updates on a given element of the system.

Although structurally similar, vector clocks and version vectors play different roles on distributed systems. Vector clocks are known to provide a view over a distributed computation, different events being identified by distinct vector clock values[1]. The role of version vectors is to detect mutual inconsistency among replicas and to determine the most recent version among two causally related replicas.

[1]In Fidge Logical Time, two events share the same clock value when representing a synchronisation event between two instances. Usually, asynchronous message passing is assumed and this does not occur.

All replicas that have seen the same updates, typically after a synchronisation procedure, share the same version vector value – see again Figure 4.1.

A well known problem of version vectors and vector clocks is that they are unbounded in size [RRP97, TRA99]. In fact, they are twice unbounded. Each integer counter can grow indefinitely and the number of identified entities can also grow unbounded.

A less known problem, which we address in this chapter, resides in the identification requirement of both version vectors and vector clocks [BM99, PST+97]. Each participating entity must be assigned a unique identifier in order to obtain a proper mapping to integer counters. In a well connected environment, it would be simple to request a unique identifier from a server or to run a distributed protocol for the generation of a unique identifier. Such protocols are not possible in the current mobile setting when subject to partitioned operation. Moreover, significant technology and research trends are pointing towards wireless ad hoc networking setups, where entities are autonomous and operate in local clusters on a proximity basis [MJK+00, BCCS98, HNI+98]. In such environments, partitioned operation is the common mode of operation and an answer to the identification problem must be sought.

In circumstances in which we can afford probabilistically unique identifiers, algorithms may resort to some form of random based ids in order to cope with replica creation under partitioned environments. Contrary to these approaches, our work does not rely on probabilistic uniqueness and assumes that guaranteed unique identifiers must be provided.

4.1.1 Fixed vs. Variable number of Replicas

Classic replication systems operate over a well defined number of replicas. Such is the case of the system depicted in Figure 4.1. The more general case of a dynamic replication system, introduces the need to accommodate replica creation and retirement. One approach would be to represent replica creation by introducing new horizontal lines and new replica identifiers in the system representation, and likewise to discontinue those lines towards the future, upon replica retirement.

The approach we follow, instead, represents all the functionality of replica creation, synchronisation and retirement by two simple constructs: replica forking and joining of replicas. Synchronisation can then be represented by joining two replicas and forking the resulting one. An example is presented in Figure 4.2.

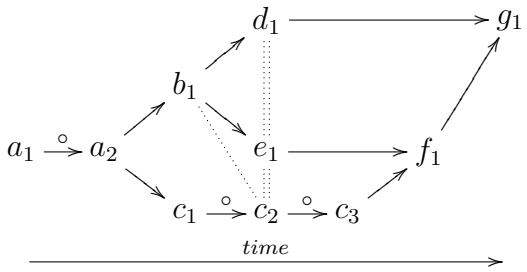


Figure 4.2: Some possible evolutions of data elements showing two frontiers of coexisting elements (denoted by single and double-dotted lines).

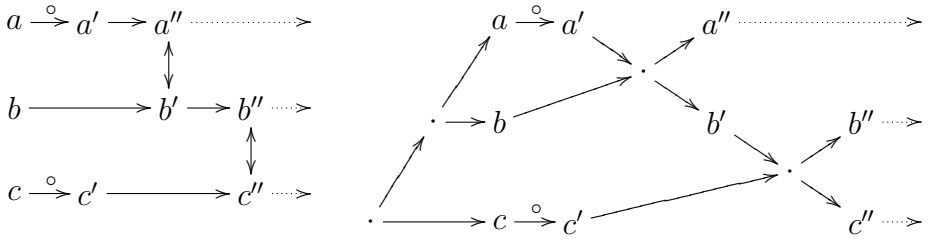


Figure 4.3: Encoding a fixed number of replicas (left) under fork-and-join dynamics (right).

This dynamic replication system is more general than the fixed one and can be used to encode the latter. In Figure 4.3 we give the intuition to this encoding by representing under fork-and-join dynamics a traditional version vector setting for three replicas, using the same names for elements in equivalent positions and omitting the name of extra elements. From this example, it is also easy to see that an equivalent mapping can be found for runs with a variable number of replicas.

4.1.2 Frontier Elements vs. All Elements

In certain circumstances, one may want to relate any two elements occurring in the distributed evolution, that is, all elements in the distributed computation are

subject to ordering. For instance, in the computation depicted in Figure 4.2, one may want to inquire how c_2 and a_1 relate and determine that a_1 is in the past of c_2. Such querying could be necessary when debugging a recorded execution of the replicated system.

In other circumstances, namely in update tracking, one may only need to relate coexisting elements, that is, only elements in the same reachable configuration. If this is the case, it wouldn't make sense to query how c_2 and a_1 relate since these elements never coexist in any arbitrary system evolution. In this sense, a reachable configuration is perceived as forming a *frontier*. Any two elements that are connected by a direct arrowed path never coexist, and consequently never belong to the same frontier of contemporaneous elements.

If we concentrate on element c_2 we can observe that, for the depicted evolution, there are two possible frontiers to which c_2 can belong. The first, represented by a single dotted line, might occur if c_1 gave place to c_2 before the bifurcation of b_1. The second frontier, double dotted, occurs if b_1's bifurcation is prior to c_2's transformation into c_3. In fact, it is possible that both frontiers occur in a particular system run.

In any case, an ordering system that targets frontier elements should have enough information to relate any two events that can occur in any possible system frontier. It is intuitive to accept that ordering of frontier elements is sufficient for version management, since only coexisting elements are subject to queries on their relation properties. We believe that this observation can have an important impact on the design of future version management techniques.

Under the distinction that we have just presented it is now clear that traditional version vectors are overly expressive: they are capable of overall ordering albeit in their application context a frontier ordering would be sufficient. One could conjecture that a compressed substitute of version vectors would be conceivable for traditional settings with fixed numbers of entities, and such substitute would not contradict Charron-Bost minimality results [CB91] (stated in the context of vector clocks but easily inferable for version vectors). This is not, however, the purpose of this article.

It is easy to conclude that classical (fixed size) version vectors are associated to frontiers of constant size, the vector dimension, while dynamic forms of version vectors, c.f. [RRP97], act on variable frontiers.

Our goal is to develop a decentralised, autonomous form of version vectors – named *version stamps* – that allows frontier ordering with autonomous creation of identifiers from any available replica. By considering frontier ordering we seek a compact solution to the identification problem that can act as an alternative to version vectors in dynamic settings.

4.1.3 Structure of the Chapter

The rest of the chapter is structured as follows. The next section introduces a model of causal histories of events, using a global view on events. Sections 3 and 4 develop the concept of version stamps and introduce a set of invariants over their structure. Section 5 establishes a functional equivalence between version stamps and causal histories, and Section 6 refines the version stamp model while keeping the equivalence. Section 7 concludes the article.

4.2 Causal Histories in Dynamic Settings

Detection of version dependencies among data elements can be constructed over a notion of causal history of update events [SM94]. In the construction of such history we assume a global view over the system in order to obtain a description that is intuitively correct. Afterwards, a version stamping system that does not rely on a global view will be constructed and proved to represent the same dependency order between elements that can be derived from the causal history.

To model causal histories we keep a mapping from element identities to sets of update events. Since we are only interested in comparing frontier elements, we only keep in the mapping the set of elements that define each frontier (thus elements that may have existed in its past are not included). This map can be seen as representing a “current configuration”.

Operations (update, fork and join) are described by transformations between configurations.

We use the traditional notation for functions: $\{a \mapsto \{x\}, b \mapsto \{y, z\}, c \mapsto \{x, z, w\}\}$ represents a function that maps elements a, b and c to sets of events; some events (like x and z) can be in the causal history of several elements.

Notation. We use $\{F; a \mapsto x, b \mapsto y\}$ to represent a function that maps a to x, b to y and that maps other elements in the domain according to function F. This notation expresses also that both a and b do not belong to the domain of F. This is useful to perform "pattern matching" over functions (Note that using $F \cup \{a \mapsto x, b \mapsto y\}$ does not imply that $x, y \notin \mathrm{dom}(F)$.). A similar notation can be used for 'pattern matching' over sets: $\{A; a, b\}$ denotes a set $A \cup \{a, b\}$ such that $a, b \notin A$.

Definition 4.2.1 *An initial configuration can be captured by* $\{a \mapsto \{\}\}$ *and represents a system with one data element. From any reachable configuration, the following transformations can occur:*

- $\{C; a \mapsto A\} \xrightarrow{\mathbf{update}(a)} \{C; a' \mapsto A \cup \{e\}\}$ *with* $e \notin \mathcal{E}(\{C; a \mapsto A\})$,
- $\{C; a \mapsto A\} \xrightarrow{\mathbf{fork}(a)} \{C; b \mapsto A, c \mapsto A\}$,
- $\{C; a \mapsto A, b \mapsto B\} \xrightarrow{\mathbf{join}(a,b)} \{C; c \mapsto A \cup B\}$,

with $\mathcal{E}(\{C\}) \doteq \bigcup\{C(i) \mid i \in \mathrm{dom}(C)\}$.

Although mapping only "current" elements, the corresponding event sets store all update events that have occurred in the causal history of each element: events are not discarded. A global view is present because each update event has a global unique identity that cannot be computed by only looking at the element being updated.

When querying the relationship between elements, according to known updates, the goal is to distinguish three possible situations: Equivalence – the same set of events; Obsolescence – all the update events and at least one more in the

dominating element; Mutual inconsistency – at least one different update event in each element. Given a configuration $\{C; a \mapsto X, b \mapsto Y\}$:

- a **equivalent to** b iff $X = Y$,
- a **obsolete relative to** b iff $X \subset Y$,
- a **inconsistent with** b iff $X \not\subseteq Y$ and $Y \not\subseteq X$.

Comparison of elements in a frontier can be deduced from the causal histories as defined above. In fact, all these situations are represented by a pre-order on the elements of a given frontier. Given a configuration C, for any two elements a, b in the domain of C, we have:

$$a \sqsubseteq_C b \iff C(a) \subseteq C(b).$$

The simplicity of this model is only possible in the presence of a global view over the set of events in the system.

4.3 Version Stamps

Our goal is to devise a stamping mechanism that can be used to infer the order between frontier elements that is induced by comparing sets of causal histories (as described above). The mechanism must not depend on any form of global view; it must work autonomously and rely only on the local information that is kept within the data elements being operated upon. An efficient use of space is also highly desirable in order to support a practical use.

We now present an informal description of version stamps. Figure 4.4 presents the example from Figure 4.2 where the version stamp corresponding to each element is shown. Each version stamp is made up of two components, which we represent as $[update \mid id]$. The *id* component acts as the element identity: it distinguishes the element from all other coexisting elements (in a frontier). The *update* component stores information about which updates are known to a given

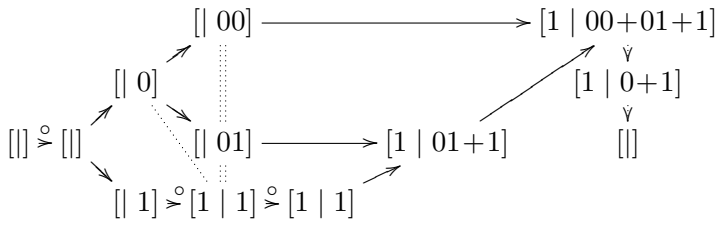


Figure 4.4: Version Stamps.

element. It avoids the use of counters and consists of a single id-like value which collects *id*'s as they were (in ancestor elements) when updates were performed. Each component is presented as a sum of binary strings.

The first two version stamps in the left show that when the frontier is only one element updates do not need to have expression on the stamps. In fact, the update operation simply copies *id* into *update*; this means that after an update, subsequent ones do not affect a version stamp. This is an example of the goal, in the design of version stamps, to discard information that is irrelevant to the comparison of coexisting elements in a frontier.

At a fork operation the *id* in the resulting stamps is recursively constructed by appending either 0 or 1 to the right of the ancestor *id*. A fork does not modify the *update* component as it does not introduce any update event (the ones tracked by the mechanism).

When a join between two elements occurs the resulting *id* is built by merging the two ancestor *id*'s. The *update* component is built likewise, merging the two ancestor *update* components; this reflects the combined knowledge of past updates.

An important property of the mechanism is the possible simplification of stamps after joins. The intuition is that a join decreases the number of elements in a frontier, leading to smaller identities being needed to distinguish them. A fork followed by a join of the resulting elements should result in an element with the original *id*. The intermediate elements *id*'s only differ in the appended 0 and 1; after being merged they are collapsed into the original *id*. (A simplification of *id* induces also a simplification of *update*.) Some analogies can be made: the simplification of minterms in boolean algebra, the collapsing of neighbour blocks

in the buddy memory allocation system [Kno65] or collecting weights in Huang's termination detection algorithm [Hua89]. Likewise, *id*'s denote non-intersecting parts of 'the whole'; their complexity adjusts dynamically, reflecting the granularity of the frontier of coexisting elements.

4.3.1 Synopsis of Formal Presentation

The locality goal of the mechanism can be seen to be met by looking at the definition of the operations (below). To prove that version stamps can be used to infer the same order as induced by causal histories, we split the presentation of version stamps and proof of correctness in several steps.

We start by presenting a non-reducing version of the mechanism, in which no simplification at joins occur, and prove several auxiliary invariants that characterise some properties of version stamps. Afterwards, we show that both causal histories and the non-reducing version of the mechanism induce the same pre-order between elements at any given frontier. To do this we must first prove a stronger result that implies the required equivalence. Finally, we present a rewriting rule on version stamps that represents the simplification after a join. We show that it preserves all previously defined invariants as well as the proved result relating causal histories to version stamps.

4.4 Version Stamps: Non-Reducing

A version stamp is a pair (u, i), respectively the *update* and the *id*. Both components share the same structure, and are members of a set N (names). We now characterise N.

Let Σ^* be the partially ordered set of all finite binary strings (sequences of $\{0, 1\}$) ordered by:

$$r \sqsubseteq s \iff r \textbf{ is a prefix of } s.$$

We have, for example, $01 \sqsubseteq 011$ and $01 \parallel 00$ (we use $\parallel$ to denote non-comparability). The null string is denoted by ϵ; it constitutes the bottom of

Σ^*: $\epsilon \sqsubseteq s$ for all strings s.

Definition 4.4.1 *N is the set of all finite antichains in Σ^*, ordered by:*

$$n_1 \sqsubseteq n_2 \iff \forall r \in n_1.\, \exists s \in n_2.\, r \sqsubseteq s.$$

For example, $\{0, 01\}$ is not a valid element of N because $0 \sqsubseteq 01$, and we have $\{00, 011\} \sqsubseteq \{000, 011, 1\}$ and $\{00, 10\} \not\sqsubseteq \{000, 011, 1\}$ as well.

As the order defined on N is the classic order in lower powerdomains [Smy78], at first sight looks like we are in the presence of a pre-order. However, N was defined in a way so that it is a partial order and not merely a pre-order. More specifically:

Proposition 4.4.2 *N is a partial order; moreover it is a join semillatice with join given by:*

$$n_1 \sqcup n_2 \doteq \{s \in n_1 \cup n_2 \mid (s \sqsubseteq r \in n_1 \cup n_2) \Rightarrow s = r\}.$$

(That is the join of two names is the set of all maximal elements in their union.)

Proof. N is isomorphic to $\mathcal{O}(\Sigma^*)$ (the down-sets of strings) ordered by inclusion, which is a complete lattice. □

Informally, the antichains in N can be seen to represent the maximal elements of down-sets, the order defined corresponds to inclusion of down-sets and the join corresponds to union of down-sets. For example, $\{00, 011\} \sqcup \{000, 01, 1\} = \{000, 011, 1\}$.

We now proceed with the definition of the first model of version stamps, in which we do not include simplification after joins. For presentation purposes, we describe the operations on version stamps using configurations that map elements to version stamps. This facilitates relating causal histories to version stamps. It is important to emphasise that this does not, however, imply that operations require a global view: the operations manipulate the version stamps of the operated upon elements, which themselves require no global view (contrary to the what happens

in causal histories, where an update operation makes use of globally unique update events). The order derived from stamps only makes use of local stamp information as well.

Definition 4.4.3 *An initial configuration can be captured by* $\{a \mapsto (\{\epsilon\}, \{\epsilon\})\}$ *and represents a system with one data element. From any reachable configuration, the following transformations can occur:*

- $\{V; a \mapsto (u, i)\} \xrightarrow{\mathbf{update}(a)} \{V; a' \mapsto (i, i)\}$,
- $\{V; a \mapsto (u, i)\} \xrightarrow{\mathbf{fork}(a)} \{V; a' \mapsto (u, i0), a'' \mapsto (u, i1)\}$ *with* $nx \doteq \{sx \mid s \in n\}, x \in \{0, 1\}$ *being the concatenation of a digit lifted to sets of strings,*
- $\{V; a \mapsto (u_a, i_a), b \mapsto (u_b, i_b)\} \xrightarrow{\mathbf{join}(a,b)} \{V; c \mapsto (u_a \sqcup u_b, i_a \sqcup i_b)\}$.

The update component simply copies the *id* into *update*; fork maintains the *update* component and appends either a 0 or a 1 to each string in the *id* component; the join operation performs joins of names for each component. It is easy to see that under the above definitions, the components in the resulting stamps are well-formed names (antichains of strings).

We now define the pre-order on the elements of a configuration V obtained from the version stamps in V, that will be used to make the correspondence with causal histories. Given a configuration V, for any two elements a, b in the domain of V, we have:

$$a \sqsubseteq_V b \iff \mathrm{fst}(V(a)) \sqsubseteq \mathrm{fst}(V(b)).$$

Towards proving a proposition that relates causal histories with version stamps we establish now some auxiliary properties of configurations of version stamps.

Invariant 4.4.4 (I_1) *In any reachable configuration* V*:* $\forall (a, (u, i)) \in V.\, u \sqsubseteq i$.

Proof. See Section 4.8. □

This invariant states that in a version stamp the *update* is always dominated by *id*. This property will ensure, on reducible version stamps models, that there

is no obsolete information on *update* when replicas converge and *id* simplifications are possible.

Invariant 4.4.5 (I_2) *In any reachable configuration V:* $\forall\{x \mapsto (u_x, i_x), y \mapsto (u_y, i_y)\} \subseteq V.\ \forall r \in i_x, s \in i_y.\ r \parallel s$.

Proof. See Section 4.8. □

This second invariant brings attention to some structural properties of the *id*'s that are present in a configuration. In a given frontier of elements each string that is present in a given *id* will be non-comparable to all other strings in the same or another *id*. Consequently, all *id*'s in a frontier are non-comparable.

Invariant 4.4.6 (I_3) *In any reachable configuration V:* $\forall\{x \mapsto (u_x, i_x), y \mapsto (u_y, i_y)\} \subseteq V.\ \forall r \in u_x.\ \{r\} \sqsubseteq i_y \Rightarrow \{r\} \sqsubseteq u_y$.

Proof. See Section 4.8. □

This invariant implies a weaker one: $\forall\{x \mapsto (u_x, i_x), y \mapsto (u_y, i_y)\} \subseteq V.\ u_x \sqsubseteq i_y \Rightarrow u_x \sqsubseteq u_y$. The pertinence of this last invariant can be illustrated by an example. Suppose two non-comparable elements $a \parallel b$ with version stamps (u_a, i_a), (u_b, i_b). If an update occurs on one of them, for instance $update(a)$, we must be sure that a (a' after update) remains non-comparable to b, and $b \sqsubseteq a'$ does not happen (recall that causal histories ensure this by using *fresh* event names on updates). Since $update(a)$ produces version stamp (i_a, i_a) then our property $u_b \sqsubseteq i_a \Rightarrow u_b \sqsubseteq u_a$ means that in order for $b \sqsubseteq a'$ to occur, then $b \sqsubseteq a$ must also occur in the first place.

4.5 Correspondence Between Causal Histories and Version Stamps

We now show that version stamps as defined above can be used to derive the pre-order between elements according to inclusion of causal histories. As we described

above, comparing elements in a configuration C of causal histories can be done according to:

$$a \sqsubseteq_C b \iff C(a) \subseteq C(b).$$

If we have a configuration V of version stamps that corresponds to C (whose version stamps are derived from the same system execution as C), being the order between elements obtained from V:

$$a \sqsubseteq_V b \iff \mathrm{fst}(V(a)) \sqsubseteq \mathrm{fst}(V(b)),$$

we want to prove that both C and V induce the same pre-order, i.e. $\sqsubseteq_C = \sqsubseteq_V$. This means we want to show that:

$$C(a) \subseteq C(b) \Leftrightarrow \mathrm{fst}(V(a)) \sqsubseteq \mathrm{fst}(V(b)).$$

It can be seen that a direct proof by induction of this equivalence fails. This failure is in itself an interesting result and can be briefly explained by the following insight: knowing how elements compare according to causal history inclusion in a given configuration is not enough to know how they will compare in the configuration obtained after performing a given operation. In other words, even though we are not interested in knowing the exact update events in causal histories, we need to know something more than just how they compare even if comparison is all we are interested in.

Technically, we need to prove a stronger equivalence, which will be used as a stronger induction hypothesis in the proof. We show then, the following stronger proposition. (We use fst and snd for the projections on the first and second components of a pair. We also use the notation $f[A]$ for the direct image of A under f, that is $f[A] = \{f(x) \mid x \in A\}$.)

Proposition 4.5.1 *Given any distributed execution with causal histories* $C_0 \longrightarrow C_1 \longrightarrow \ldots \longrightarrow C_k$ *and with version stamps* $V_0 \longrightarrow V_1 \longrightarrow \ldots \longrightarrow V_k$*, it is true that* $\mathrm{dom}(C_k) = \mathrm{dom}(V_k)$ *and* $C_k(x) \subseteq \bigcup C_k[S] \Leftrightarrow \mathrm{fst}(V_k(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V_k[S]]$*, for all* $x \in \mathrm{dom}(C_k)$ *and* $\emptyset \subset S \subseteq \mathrm{dom}(C_k)$.

Proof. See Section 4.8. □

From the previous proposition, the result we want to show follows, as stated by:

Corollary 4.5.2 *Given any distributed execution with causal histories* $C_0 \longrightarrow C_1 \longrightarrow \ldots \longrightarrow C_k$ *and with version stamps* $V_0 \longrightarrow V_1 \longrightarrow \ldots \longrightarrow V_k$*, it is true that* $\mathrm{dom}(C_k) = \mathrm{dom}(V_k)$ *and* $C_k(x) \subseteq C_k(y) \Leftrightarrow \mathrm{fst}(V_k(x)) \sqsubseteq \mathrm{fst}(V_k(y))$*, for all* x, y *in* $\mathrm{dom}(C_k)$*.*

Proof. Substitute S by $\{y\}$ in the previous proposition. □

4.6 Simplifying Version Stamps upon Joins

We now describe a rewriting rule that can be applied to a version stamp and perform the simplifications that have been informally introduced in Figure 4.4. Such simplifications reflect, as already discussed, the dynamic adaptation of *id*'s to the 'shape' of the frontier. This simplification is essential towards obtaining a realistic implementation, by minimising the space requirements of version stamps.

The simplification of a version stamp that results from a join is attempted by repeatedly applying the following rewriting rule until it is no longer possible to apply it:

$$(u, \{i; s0, s1\}) \xrightarrow{\alpha} (u', \{i; s\}),$$

with

$$u' = \begin{cases} u \setminus \{s0, s1\} \cup \{s\} & \text{if } s0 \in u \text{ or } s1 \in u, \\ u & \text{otherwise.} \end{cases}$$

One property of a rewriting $(u, i) \xrightarrow{\alpha} (u', i')$ that follows trivially from the order on names is that $u' \sqsubseteq u$ and $i' \sqsubseteq i$. As the order on names is well-founded (there are no infinite descending chains of names), only a finite number of rewritings

can be applied to a stamp. It is also easy to see that the rewriting is confluent. Therefore, a stamp can be rewritten into a unique normal form.

We omit the proof of confluence as it is intuitive and concentrate on the correctness of the transformation. For that we need to show that applying a rewriting $(u, \{i; s0, s1\}) \xrightarrow{\alpha}$
$(u', \{i; s\})$ to a version stamp in a configuration V results in a configuration V' where: the rewritten version stamp consists of two wellformed names (antichains), the invariants I_1, I_2, I_3 are maintained, and the relation from $\mathrm{dom}(V)$ to $\mathcal{P}(\mathrm{dom}(V))$ expressed by

$$R(V) \doteq \{(x, S) \mid \mathrm{fst}(V(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V[S]]\}$$

is the same in V', i.e. $R(V) = R(V')$.

Wellformedness of u' and $\{i; s\}$. Regarding $\{i; s\}$, as $\{i; s0, s1\}$ is an antichain, we have for every $r \in i$ that $r \parallel s0$ and $r \parallel s1$; therefore $r \parallel s$, which means that $\{i; s\}$ is also an antichain. Regarding u', if neither $s0$ nor $s1$ belong to u, then $u' = u$. Otherwise, we have for every $r \in u \setminus \{s0, s1\}$ that: $s0 \not\sqsubseteq r$ and $s1 \not\sqsubseteq r$ (because $u \sqsubseteq \{i; s0, s1\}$), and $r \not\sqsubseteq s$ (because u is an antichain); therefore, $r \parallel s$, which means that $u \setminus \{s0, s1\} \cup \{s\}$ is an antichain.

Invariant I_1. This is a local invariant on each stamp; it suffices to show that $u' \sqsubseteq \{i; s\}$. If neither $s0$ nor $s1$ belong to u, then $u' = u \sqsubseteq \{i; s\}$ (as $u \sqsubseteq \{i; s0, s1\}$). Otherwise, it is also trivial that $u' = u \setminus \{so, s1\} \cup \{s\} \sqsubseteq \{i; s\}$, for the same reason.

Invariant I_2. This invariant involves pairs of stamps; it suffices to consider the cases where the rewritten stamp is involved. For any other stamp (u_x, i_x) in V and string $r \in i_x$, due to Invariant I_2 on V we have: $r \parallel s0$, $r \parallel s1$, therefore $r \parallel s$; and also $r \parallel t$ for all $t \in i$; therefore $r \parallel v$ for all strings $v \in \{i; s\}$.

Invariant I_3. The invariant involves expressions of the form $\{r\} \sqsubseteq i_y \Rightarrow \{r\} \sqsubseteq u_y$, for stamps (u_x, i_x), (u_y, i_y), and $r \in u_x$. As for the previous invariant, it suffices to consider the cases where the rewritten stamp is involved:

$(u', \{i; s\}) = (u_y, i_y)$**.** Suppose $\{r\} \sqsubseteq \{i; s\}$; then, $\{r\} \sqsubseteq \{i; s0, s1\}$ and by I_3 on configuration V $\{r\} \sqsubseteq u$. If neither $s0$ nor $s1$ belong to u, then $u' = u$ and $\{r\} \sqsubseteq u'$. Otherwise, as $\{r\} \sqsubseteq \{i; s\}$, we have $r \neq s0$ and $r \neq s1$; therefore, $\{r\} \sqsubseteq u \setminus \{so, s1\} \cup \{s\} = u'$.

$(u', \{i; s\}) = (u_x, i_x)$**.** Suppose $\{r\} \sqsubseteq i_y$ with $r \in u'$. If neither $s0$ nor $s1$ belong to u, then $u' = u$, $r \in u$; therefore, by I_3 on V, we have $\{r\} \sqsubseteq u_y$. Otherwise, in which case $u' = u \setminus \{so, s1\} \cup \{s\}$, we have $r \neq s0$ and $r \neq s1$; also $r \neq s$, otherwise we would have $\{s\} \sqsubseteq i_y$, impossible by I_2 on V; therefore $r \in u$ and by I_3 on V we have $\{r\} \sqsubseteq u_y$.

Preservation of R. We prove that applying a rewriting $(u, \{i; s0, s1\}) \xrightarrow{\alpha} (u', \{i; s\})$ to a version stamp in a configuration V results in a configuration V' so that $R(V) = R(V')$. First we show that $x\ R(V)\ S \Rightarrow x\ R(V')\ S$. Suppose $x\ R(V)\ S$, i.e. $\mathrm{fst}(V(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V[S]]$. We must consider two cases:

rewriting of $V(x)$. If $x \in S$ then $x\ R(V')\ S$ holds trivially; if $x \notin S$, as $u' \sqsubseteq u$, we have $u' \sqsubseteq u \sqsubseteq \bigsqcup \mathrm{fst}[V[S]] = \bigsqcup \mathrm{fst}[V'[S]]$.

rewriting of $V(y)$ with $y \in S$. The case $x = y$ is trivial (and already covered above). Otherwise $x \neq y$; let $Z = S \setminus \{y\}$; we have $V|Z = V'|Z$ and also $V(x) = V'(x)$; therefore $\mathrm{fst}(V'(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V'[Z]] \sqcup u$. If $s0 \notin u$ and $s1 \notin u$ we have $u' = u$ and $x\ R(V')\ S$ holds trivially. Otherwise, in which case $u' = u \setminus \{s0, s1\} \cup \{s\}$, due to I_1 and I_2 (and $x \neq y$) we have that $s0$ and $s1$ do not belong to $\mathrm{fst}(V(x))$; therefore the inequality above still holds replacing u by u', and thus we have $x\ R(V')\ S$.

Now we show that $x\ R(V')\ S \Rightarrow x\ R(V)\ S$. Suppose $x\ R(V')\ S$, i.e. $\mathrm{fst}(V'(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V'[S]]$. We must consider two cases:

rewriting of $V(y)$ with $y \in S$. The case $x = y$ is trivial; if $x \neq y$, given that $u' \sqsubseteq u$, we have $\text{fst}(V(x)) = \text{fst}(V'(x)) \sqsubseteq \bigsqcup \text{fst}[V'[S]] \sqsubseteq \bigsqcup \text{fst}[V[S]]$.

rewriting of $V(x)$. The case $x \in S$ is trivial; in the case $x \notin S$, we have that no string in $\text{fst}[V'[S]]$ is greater than or equal to s, otherwise, due to I_1 there would exist a string r in $\text{snd}[V'[S]]$ such that $s \sqsubseteq r$, something impossible because, due to I_2, no string in $\text{snd}[V'[S]]$ can be comparable to s Therefore, $s \notin u'$ which means we are in the case where $u' = u$ and so we have also $\text{fst}(V(x)) \sqsubseteq \bigsqcup \text{fst}[V[S]]$.

4.7 Conclusions

Both version vectors and vector clocks rely on the availability of identifiers that can support their ordering technique. We have argued that operation under partitioned operation and mobility prevents the use of traditional techniques for unique identifier generation, and that these operation modes are already common and call for appropriate solutions. Additionally, data management under these operation modes is mostly based on optimistic techniques and therefore requires robust dependency tracking solutions.

In this article we addressed the identification problem in the context of data dependency tracking. In order to achieve this goal we had to distinguish the ordering of elements in a frontier from the ordering of any two elements in a system run, thus contributing to the clarification of the role of version vectors. This distinction, together with the presence of the identification problem, raises a set of research lines, one of which was developed in the article. The other lines concern the design of decentralised vector clocks, by exploring autonomous identifiers on overall ordering, and the search for a more compact (possibly bound) form of version vectors on settings with fixed identifiers and frontier ordering.

We have developed a model of causal histories that is adapted to dynamic settings exhibiting autonomous interaction. We presented a version stamping mechanism that only relies on information that is locally available, overcoming the need for a global view. Finally, we established and proved a correspondence which

states that the relation between any two given elements in a frontier, according to inclusion of causal histories, can be computed by their version stamps.

Version stamps, having solved the autonomous identification problem while addressing frontier ordering, provide an adequate dependency tracking mechanism that operates in scenarios where this functionality was not available.

The presented version stamp mechanism has been implemented in the PANASYNC project[2] [ABF00]. This project is an application of version stamps to file replication, providing a set of tools for dependency tracking on single file copies. The project provides a C++ STL based library implementing version stamps.

4.8 Proof of Invariants and Main Proposition

Invariant 4.4. (I_1) *In any reachable configuration V:* $\forall a \mapsto (u,i) \in V.\, u \sqsubseteq i$.

Proof. The proof is by induction. In the base case we have $V_0 = \{a \mapsto (\{\epsilon\},\{\epsilon\})\}$. The invariant holds since $\epsilon \sqsubseteq \epsilon$. The inductive step will suppose that our invariant I_1 holds on a given environment V and check for its validity under V' that results from applying any of the operations update, fork, join.

update(a)**.** From the definition of the operation we have a new element a' with $V'(a') = (i,i)$. Since $i \sqsubseteq i$ the invariant holds.

fork(a)**.** From the definition we have $V(a) = (u,i)$ and $u \sqsubseteq i$ by induction hypothesis. In V' we have $V'(a') = (u, i0)$ and $V'(a'') = (u, i1)$. We verify that $u \sqsubseteq i0$ holds by checking the definition of name concatenation together with hypothesis $u \sqsubseteq i$. The same applies to $u \sqsubseteq i1$.

join(a,b)**.** From the definition and by induction hypothesis we have in V, $u_a \sqsubseteq i_a$ and $u_b \sqsubseteq i_b$. We must infer in V' that $u_a \sqcup u_b \sqsubseteq i_a \sqcup i_b$. This proposition can be

[2] `http://sourceforge.net/projects/panasync`.

directly deduced from the above two hypothesis due to the properties of the join semilattice. □

Invariant 4.5. (I_2) *In any reachable configuration V:* $\forall\{x \mapsto (u_x, i_x), y \mapsto (u_y, i_y)\} \subseteq V.\ \forall r \in i_x, s \in i_y.\ r \parallel s$.

Proof. The proof is again by induction using the same structure as above. In the base case there are no distinct i_x, i_y so the invariant holds trivially.

update(a)**.** Under this operation knowing that $V(a) = (u, i)$ holds, $V'(a') = (i, i)$ will hold in V'. Since both id's are i the invariant, true in V by hypothesis, is preserved.

fork(a)**.** From the definition we have $V(a) = (u, i)$ and from induction hypothesis, all $i_x \neq i$ in V exhibit $\forall r \in i_x, s \in i.\ r \parallel s$. In V' we have $V'(a') = (u, i0), V'(a'') = (u, i1)$. From iterated concatenation, on fork definition, we infer $t \parallel v \Rightarrow t0 \parallel v$ (as well as $t1 \parallel v$) for any $t, v \in S$. This and the induction hypothesis proves $\forall r \in i_x, s \in i0\ r \parallel s$, with identical reasoning for $i1$. Considering $i0$ and $i1$, $\forall r \in i1, s \in i0.\ r \parallel s$ results from iterated concatenation.

join(a, b)**.** From the definition we have $V(a) = (u_a, i_a), V(b) = (u_b, i_b)$ and from induction hypothesis, all $i_x \neq i_a \neq i_b$ exhibit $\forall r \in i_x, t \in i_a, v \in i_b.\ r \parallel t \wedge r \parallel v \wedge t \parallel v$. Consequently $\forall r \in i_x, s \in (i_a \cup i_b).\ r \parallel s$. In V' we have $V'(c) = (u_a \sqcup u_b, i_a \sqcup i_b)$. Since $i_a \sqcup i_b \subseteq i_a \cup i_b$ (in fact, here $i_a \sqcup i_b = i_a \cup i_b$) we reach $\forall r \in i_x, s \in (i_a \sqcup i_b).\ r \parallel s$. □

Invariant 4.6. (I_3) *In any reachable configuration V:* $\forall\{x \mapsto (u_x, i_x), y \mapsto (u_y, i_y)\} \subseteq V.\ \forall r \in u_x.\ \{r\} \sqsubseteq i_y \Rightarrow \{r\} \sqsubseteq u_y$.

Proof. This proof is by induction, and for each operation the invariant validity is inferred. In this proof, notation of the form $a = b \sqsubseteq c$ signifies $a = b$ and $b \sqsubseteq c$. The invariant $\{r\} \sqsubseteq i_y \Rightarrow \{r\} \sqsubseteq u_y$ is checked by verifying that either $\{r\} \not\sqsubseteq i_y$

or $\{r\} \sqsubseteq i_y \wedge \{r\} \sqsubseteq u_y$ holds. In the base case, there is only one element and the invariant holds trivially.

update(a)**.** From $V(a) = (u, i)$ we have $V'(a') = (i, i)$. Suppose any two stamps $(u'_x, i'_x), (u'_y, i'_y)$ in V' and $\forall r' \in u'_x$. If $(u'_x, i'_x) \neq (i, i)$ and $(u'_y, i'_y) \neq (i, i)$ then $(u_x, i_x), (u_y, i_y)$ occur in V and the invariant holds from V by induction hypothesis. Otherwise we must consider two cases:

- $(c'_x, i'_x) = (i, i)$ in which case $\{r'\} \not\sqsubseteq i'_y$ since $c'_x = i \neq i'_y$ and (I_2).
- $(c'_y, i'_y) = (i, i)$ in which case $(u_y, i_y) = (u, i)$ and either: $\{r\} \not\sqsubseteq i_y$ held in V, leading in V' to $\{r'\} \not\sqsubseteq i'_y$; or $\{r\} \sqsubseteq i_y \wedge \{r\} \sqsubseteq u_y$ held in V and becomes induction hypothesis. In V', $\{r'\} \sqsubseteq i'_y$ still holds since $i'_y = i_y$, and $\{r'\} \sqsubseteq c'_y$ can be inferred, since $u'_x = u_x$ and $\{r\} \sqsubseteq u_y = u \sqsubseteq i = u'_y$.

fork(a)**.** From $V(a) = (u, i)$ we have $V'(a') = (u, i0), V'(a'') = (u, i1)$. Suppose any two stamps $(u'_x, i'_x), (u'_y, i'_y)$ in V' and $\forall r' \in u'_x$. If $(u'_x, i'_x) \neq (u, i0), (u'_y, i'_y) \neq (u, i1)$ identical $(u_x, i_x), (u_y, i_y)$ occurred in V and the invariant is kept. Otherwise consider three cases (the other cases are obtained by swapping 0 and 1):

- $(u'_x, i'_x) = (u, i0), (u'_y, i'_y) \neq (u, i1)$ in which case there was in V an identical (u_y, i_y) and for $(u_x, i_x) = (c, i)$ we had either: $\{r\} \not\sqsubseteq i_y$ and consequently $\{r'\} \not\sqsubseteq i'_y$ since $u'_x = u = u_x$; or we had $\{r\} \sqsubseteq i_y \wedge \{r\} \sqsubseteq u_y$, becoming induction hypothesis. In V', $\{r'\} \sqsubseteq i'_y$ still holds, and $\{r'\} \sqsubseteq u'_y$ can be inferred, since $u'_x = u_x$ and $\{r\} \sqsubseteq u_y = u'_y$.
- $(u'_x, i'_x) \neq (u, i0), (u'_y, i'_y) = (u, i1)$ in which case there was in V an identical (u_x, i_x) and for $(u_y, i_y) = (u, i)$ we had either: $\{r\} \not\sqsubseteq i_y = i$ and consequently $\{r'\} \not\sqsubseteq i'_y = i1$ since iterated concatenation cannot revert the $\not\sqsubseteq$ relation; or we had $\{r\} \sqsubseteq i_y \wedge \{r\} \sqsubseteq u_y$, now becoming induction hypothesis. Again, in V', $\{r'\} \sqsubseteq i'_y$ will hold since $u'_x = u_x$ and $\{r\} \sqsubseteq i_y = i \sqsubseteq i1 = i'_y$. $\{r'\} \sqsubseteq u'_y$ also holds, from $u'_x = u_x$ and $\{r\} \sqsubseteq u_y = u = u'_y$.
- $(u'_x, i'_x) = (u, i0), (u'_y, i'_y) = (u, i1)$ in which case we known that $\{r'\} \sqsubseteq i'_y$ holds, since $u'_x = u \sqsubseteq i \sqsubseteq i1 = i'_y$. $\{r'\} \sqsubseteq u'_y$ also holds trivially since $u'_x = u = u'_y$.

join(a, b)**.** From $V(a) = (u_a, i_a), V(b) = (u_b, i_b)$ we have $V'(c) = (u_a \sqcup u_b, i_a \sqcup i_b)$. Suppose any two stamps $(u'_x, i'_x), (u'_y, i'_y)$ in V' and $\forall r' \in u'_x$. If none of these stamps matches $(u_a \sqcup u_b, i_a \sqcup i_b)$, identical $(u_x, i_x), (u_y, i_y)$ occurred in V and the invariant is kept. Otherwise consider two cases:

$(u'_x, i'_x) = (u_a \sqcup u_b, i_a \sqcup i_b)$ in which case $\{r'\} \not\sqsubseteq i'_y$ will hold in V' if there is a $v = r'$ in either u_a or u_b such that $\{v\} \not\sqsubseteq i_y = i'_y$. Otherwise, $\{v\} \sqsubseteq i_y \wedge \{u\} \sqsubseteq u_y$ become induction hypothesis, and both $\{r'\} \sqsubseteq i'_y = i_y$ and $\{r'\} \sqsubseteq u'_y = u_y$ hold in V'.

$(u'_y, i'_y) = (u_a \sqcup u_b, i_a \sqcup i_b)$ in which case $\{r'\} \not\sqsubseteq i'_y = i_a \sqcup i_b$ will hold in V' if in V $\{r\} \not\sqsubseteq i_a$ and $\{r\} \not\sqsubseteq i_b$. Otherwise, one or both of $\{r\} \sqsubseteq i_a \wedge \{r\} \sqsubseteq u_a$ and $\{r\} \sqsubseteq i_b \wedge \{r\} \sqsubseteq u_b$ hold in V and become induction hypothesis. In such case, $\{r'\} \sqsubseteq i'_y = i_a \sqcup i_b$ can be inferred, since $\{r\} \sqsubseteq i_a$ (or i_b) and $i_a \sqsubseteq i_a \sqcup i_b$. Similarly, $\{r'\} \sqsubseteq u'_y = u_a \sqcup u_b$ is inferred under this hypothesis.

□

Proposition 5.1 *Given any distributed execution with causal histories* $C_0 \longrightarrow C_1 \longrightarrow \ldots \longrightarrow C_k$ *and with version stamps* $V_0 \longrightarrow V_1 \longrightarrow \ldots \longrightarrow V_k$*, it is true that* $\mathrm{dom}(C_k) = \mathrm{dom}(V_k)$ *and* $C_k(x) \subseteq \bigcup C_k[S] \Leftrightarrow \mathrm{fst}(V_k(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V_k[S]]$*, for all* $x \in \mathrm{dom}(C_k)$ *and* $\emptyset \subset S \subseteq \mathrm{dom}(C_k)$*.*

Proof. The proof is by induction. In the base case we have $C_0 = \{a \mapsto\}$ for some a and $V_0 = \{a \mapsto (\{\epsilon\}, \{\epsilon\})\}$; both domains are equal ($\{a\}$); and the equivalence holds trivially.

The inductive step for domain equality is trivial, given the definition of each operation, which transforms each domain in the same way (e.g. compare Definitions 4.2.1 and 4.4.3 regarding the update operation). The inductive step for the family of equivalencies consists of, assuming that the equivalencies $C(x) \subseteq \bigcup C[S] \Leftrightarrow \mathrm{fst}(V(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V[S]]$ hold for two given environments C and V, they will hold for the environments C', V' that result from applying any of the operations **update**, **fork**, **join** to C and V, i.e. $C'(x) \subseteq \bigcup C'[S] \Leftrightarrow \mathrm{fst}(V'(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V'[S]]$ will

hold. For each operation we prove the equivalence by showing implication in both directions.

update(a)**.** From the definition of the operation, we have $C'(b) = C(a) \cup \{e\}$ for some b, e; $V(a) = (u, i)$ for some (u, i); $V'(b) = (i, i)$. First we prove ($\Rightarrow$). Assume $C'(x) \subseteq \bigcup C'[S]$. We must consider two cases:

$b \notin S$ in which case $e \notin \bigcup C'[S]$ and also $x \neq b$ (otherwise we would have $e \in C'(x)$ which would contradict the assumption $C'(x) \subseteq \bigcup C'[S]$); therefore $C'(x) = C(x)$. As also $C'|S = C|S$ (as $b \notin S$), we have $C(x) \subseteq \bigcup C[S]$, and by the induction hypothesis $\mathrm{fst}(V(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V[S]]$. As also $V'(x) = V(x)$ and $V'|S = V|S$, it follows trivially that $\mathrm{fst}(V'(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V'[S]]$.

$b \in S$ The case $x = b$ is trivial. In the case $x \neq b$, we have $C'(x) = C(x)$. Let $T = S \setminus \{b\}$; we have $C'|T = C|T$. As $C'(b) = C(a) \cup \{e\}$, the assumption becomes $C'(x) \subseteq \bigcup C'[T] \cup C(a) \cup \{e\}$; therefore $C(x) \subseteq \bigcup C[T] \cup C(a)$ (as $e \notin C'(x)$). By the induction hypothesis, $\mathrm{fst}(V(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V[T]] \sqcup \mathrm{fst}(V(a))$. As $V(a) = (u, i)$, $V'(b) = (i, i)$ and $u \sqsubseteq i$ from Invariant I_1, and also $V(x) = V'(x)$ and $V'|T = V|T$, we obtain $\mathrm{fst}(V'(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V'[S]]$.

Now we prove ($\Leftarrow$). Assume $\mathrm{fst}(V'(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V'[S]]$. Again we must consider two cases:

$b \notin S$ in which case we have also $x \neq b$; otherwise we would have $V'(x) = V'(b) = (i, i)$, and there is no $y = (u_y, i_y) \in S$ such that $V(b) \sqsubseteq V'(y)$ (by Invariant I2 $i_y \parallel i$ and by I1 $u_y \sqsubseteq i_y$, thus $u_y \parallel i$), which would contradict the induction hypothesis. Therefore $C'(x) = C(x)$, $C'|S = C|S$, $V'(x) = V(x)$ and $V'|S = V|S$ and by the induction hypothesis it follows trivially that $C'(x) \subseteq \bigcup C'[S]$.

$b \in S$ The case $x = b$ is trivial. Considering $x \neq b$, let $T = S \setminus \{b\}$. We have $\mathrm{fst}(V'(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V'[S]] = \bigsqcup \mathrm{fst}[V'[T]] \sqcup \mathrm{fst}(V'(b))$, and $V(x) = V'(x)$, $V'|T = V|T$. It follows $\mathrm{fst}(V(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V[T]] \sqcup \mathrm{fst}(V(a))$; otherwise we would have $s \in \mathrm{fst}(V(x))$ such that $\{s\} \sqsubseteq i$, $\{s\} \not\sqsubseteq c$ in which case Invariant

I3 would not hold. By induction hypothesis, $C'(x) = C(x) \subseteq \bigcup C[T] \cup C(a)$ and since $C'(b) = C(a) \cup \{e\}$, it follows $C'(x) \subseteq \bigcup C'[T] \cup C'(b) = \bigcup C'[S]$.

fork(a)**.** This case is trivial as from the definitions we have that both causal histories and update components are preserved in this operation.

join(a, b)**.** From the definition of the operation, we have $C'(c) = C(a) \cup C(b)$, for some c. First we prove ($\Rightarrow$). Assume $C'(x) \subseteq \bigcup C'[S]$. We must consider two cases:

$c \notin S$ in which case $C'|S = C|S$, and $V'|S = V|S$. If $x \neq c$, we have also $C'(x) = C(x)$ and $V'(x) = V(x)$; using the induction hypothesis, $\text{fst}(V'(x)) \sqsubseteq \bigsqcup \text{fst}[V'[S]]$ follows trivially. If $x = c$, then $C'(x) = C(a) \cup C(b) \subseteq \bigcup C'[S] = \bigcup C[S]$. From the induction hypothesis, we have both $\text{fst}(V(a)) \sqsubseteq \bigsqcup \text{fst}[V[S]]$ and $\text{fst}(V(b)) \sqsubseteq \bigsqcup \text{fst}[V[S]]$. Therefore, $\text{fst}(V(c)) = \text{fst}(V(a)) \sqcup \text{fst}(V(b)) \sqsubseteq \bigsqcup \text{fst}[V[S]]$.

$c \in S$ The case $x = c$ is trivial. In the case $x \neq c$, let $T = S \setminus \{c\}$. We have $C'(x) = C(x)$ and $C'|T = C|T$. From the assumption, as $C'(c) = C(a) \cup C(b)$, we obtain $C(x) \subseteq \bigcup C[T] \cup C(a) \cup C(b)$; By the induction hypothesis: $\text{fst}(V(x)) \sqsubseteq \bigsqcup \text{fst}[V[T]] \sqcup \text{fst}(V(a)) \sqcup \text{fst}(V(b))$. As also $V(x) = V'(x)$, $V'|T = V|T$, and $\text{fst}(V'(c)) = \text{fst}(V(a)) \sqcup \text{fst}(V(b))$, it follows that $\text{fst}(V'(c)) \sqsubseteq \bigsqcup \text{fst}[V'[S]]$.

Now we prove ($\Leftarrow$). Assume $\text{fst}(V'(x)) \sqsubseteq \bigsqcup \text{fst}[V'[S]]$. Again we must consider two cases:

$c \notin S$ in which case $V'|S = V|S$ and $C'|S = C|S$. If $x \neq c$, we have $V'(x) = V(x)$ and $C'(x) = C(x)$; using the induction hypothesis, $C'(x) \sqsubseteq \bigcup C'[S]$ follows trivially. If $x = c$ we have $\text{fst}(V'(x)) = \text{fst}(V(a)) \sqcup \text{fst}(V(b)) \sqsubseteq \bigsqcup \text{fst}[V'[S]] = \bigsqcup \text{fst}[V[S]]$. From the induction hypothesis, we have both $C(a) \subseteq \bigcup C[S]$ and $C(b) \subseteq \bigcup C[S]$. Therefore, $C'(c) = C(a) \cup C(b) \subseteq \bigcup C[S] = \bigcup C'[S]$.

$c \in S$ The case $x = c$ is trivial. In the case $x \neq c$, let $T = S \setminus \{c\}$. We have $\mathrm{fst}(V'(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V'[S]] = \bigsqcup \mathrm{fst}[V'[T]] \sqcup \mathrm{fst}(V'(c))$, and $V(x) = V'(x)$, $V'|T = V|T$. It follows $\mathrm{fst}(V(x)) \sqsubseteq \bigsqcup \mathrm{fst}[V[T]] \sqcup \mathrm{fst}(V(a)) \sqcup \mathrm{fst}(V(b))$, and by the induction hypothesis, $C'(x) = C(x) \subseteq \bigcup C[T] \cup C(a) \cup C(b) = \bigcup C[T] \cup C'(c) = \bigcup C'[S]$.

□

Chapter 5

Improving on Version Stamps

Optimistic distributed systems often rely on version vectors or their variants in order to track updates on replicated objects. Some of these mechanisms rely on some form of global configuration or distributed naming protocol in order to assign unique identifiers to each replica. These approaches are incompatible with replica creation under arbitrary partitions, a typical operation mode in mobile or poorly connected environments. Other mechanisms assign unique identifiers relying on statistical correctness. In previous work we have introduced an update tracking mechanism that overcomes these limitations. This chapter presents results from recent experimentation, that brought to surface a particular pattern of operation that results in an unforeseen, unlimited growth in space consumption. We also describe informally a new update tracking mechanism that does not exhibit this pathological growth while providing guaranteed unique identifiers for a dynamic number of replicas under arbitrary partitions and the same functionality of version vectors.

5.1 Introduction

Tracking update dependencies on optimistic replication systems often resorts to the use of version vectors [PPR+83] or some of its variants [SS05]. These mechanisms have been devised and have been successfully supporting traditional sce-

narios with a fixed number of replicas. Extensions to version vectors have been proposed [RRP97] in order to accommodate a variable number of replicas. However, when trying to cope with the problem of replica identification there is an implicit assumption of global configuration or of a well connected environment in which a distributed naming protocol can be run and replica retirement detected. These assumptions are incompatible with replica creation (and retirement) on distributed systems subject to arbitrary partitions, a typical mode of operation of mobile or poorly connected environments. Other approaches tackle this identification problem relying on statistical correctness [KWK03]. These approaches, not only may lead to occasional errors (which, even very rare, may be unacceptable), but also lead to large identifiers.

Previous work of the present authors [ABF02b] introduced the Version Stamp mechanism, a form of decentralised version vector overcoming these limitations. It enables autonomous identity management, update tracking and comparison, solely relying in local or pair-wise knowledge. This confinement is possible because: its structure allows a local management of the identity namespace (both local generation of identifiers when forking replicas and merging of identifiers when joining replicas); and the information about updates to replicas is based on the identity namespace in such a way as to allow global comparisons. This structure was devised in such a way that it should naturally grow and collapse as replicas are created and merged in the system.

Version Stamps have been employed in the Panasync [ABF00] decentralised file replication system. Recent experimentation, however, brought to surface a particular pattern of operation that, when repeatedly applied, leads to an unnecessary unbounded growth of its structure.

This chapter identifies this particular pattern of operation and illustrates how the version stamp structure degenerates under its occurrence. It also proposes a new version tracking mechanism – inspired by recent insights on autonomous identity management – that does not exhibit this unnecessary structural growth.

5.2 Version Stamps

Version stamps were devised in order to track update dependencies across a set of replicas in a mobile or poorly connected environment. In this setting, autonomous creation of replicas and pair-wise reasoning over update dependencies are crucial requirements. Operations on version stamps cannot depend on a global view of the system and thus they rely exclusively on local knowledge of each replica.

The structure of a version stamp is made of an *identity* and an *update* component. The identity component distinguishes each replica from all coexisting ones, in any possible configuration. It is also used as an available namespace from which new identities can be generated. This identity generated is achieved by namespace division as described below. The update component records "when" (in which state) changes were applied to a replica. It consists of a single identity-like value collected from the identity of its ancestor when the update was performed.

Three operations are provided: a **fork** operation supports the creation of new replicas whose state is cloned from the original; a **join** operation supports the merging of two replicas keeping one and retiring the other; and an **update** operation accounts for changes on the state of a replica.

An update operation simply copies the *identity* to the *update* component. This means that after an update, subsequent ones do not affect a version stamp. This is an example of the goal, in the design of version stamps, to discard information that is irrelevant to the comparison of coexisting elements in a configuration.

At a fork operation the *identity* of the resulting version stamps is recursively constructed by appending either '0' or '1' to the right of each component of the ancestor *identity*. A fork does not modify the *update* component as it does not introduce any update event (the ones tracked by the mechanism): it simply copies the *update* component to the new version stamps. Regarding identity management, the transformation applied to the identity component can be perceived as the subdivision of the namespace available to a particular replica. Although a local operation, the resulting namespaces are guaranteed to be globally unique and thus distinguish the two new replicas from all the others in the current configuration.

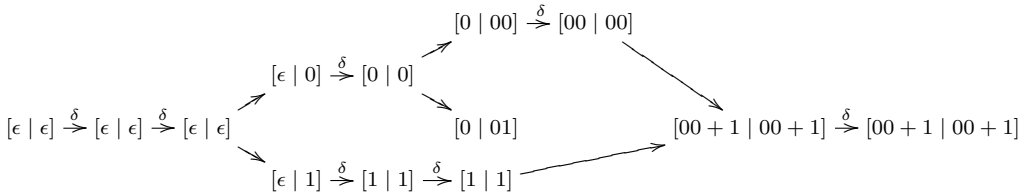


Figure 5.1: A set of partially ordered events with version stamps.

When a join between two elements occur the resulting *identity* is built by merging the two ancestors *identity* components. The *update* component is built likewise, merging the two ancestor *update* components; this reflects the combined knowledge of past updates. Upon the join operation, the resulting identity namespace can be perceived as the union of the two ancestors namespaces. This resulting namespace can then be recursively collapsed each time sibling identity namespaces are present (namespaces that have been previously split upon a fork operation). Since the *identity* component always *dominates* the *update* component (which records information regarding past state changes), this simplification propagates to the *update* component. Ultimately, joining every coexisting replica leads to a completely collapsed version stamp, bringing its structure to its initial value, that is, two empty sets. This division and collapsing feature is an intended design goal of the version stamp mechanism.

Figure 5.1 shows an example of the version stamp mechanism in action on a replicated system. In this example a version stamp is represented by an [*update* | *identity*] pair, the ϵ denotes an empty set and the δ denotes a local event of state change. Though not shown in this example, as stated above, joining the two remaining replicas would completely collapse the resulting version stamp. A detailed and formal description of Version Stamps including its proof of correctness can be found in [ABF02b].

5.2.1 Pollution of the Namespace

Exercising the version stamp mechanism, we have observed an undesired growth of version stamps under a simple pattern of operation. This problem is illustrated in Figure 5.2, which shows the identity component in a scenario where three replicas

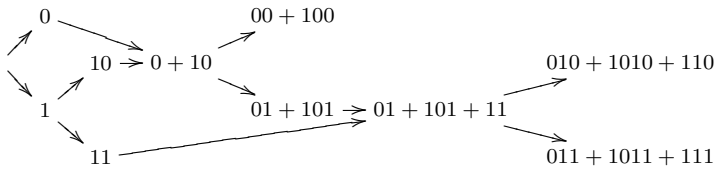


Figure 5.2: Pollution in the identity component of version stamps.

are created and then we repeat a pattern in which two of them are joined and forked again, while alternating replicas.

Although we end up with only three replicas, the identity components are much more complex than in the configuration after the first two forks (with the same number of replicas). In this scenario, the Version Stamp mechanism leads to a overly refined namespace which cannot be simplified upon these interleaving joins.

This growth gets worse every time this operation pattern occurs and recent experimentation does indicate that this can be a fairly common case in practical usage scenarios. Furthermore, when an update occurs, the identity component is copied to the update component, thus aggravating this problem.

This degeneration of the namespace does not imply that the version stamp mechanism is incorrect but that it may consume an unreasonable amount of space. As a result, this growth pattern of version vectors can severely affect its practical application.

5.3 Dynamic Map Clocks

Dynamic Map Clocks imports from version stamps the basic features that support autonomous identity management. Noticing that the lack of counters, on version stamps, impose important restrictions on identity management that ultimately contributed to the identified growth problem, dynamic map clocks will combine the use of counters with a more flexible identity management scheme.

The important property that rules identity management for update tracking is the allocation to each replica of at least one identity that is exclusive to that replica in a given moment. When an update needs registering in a given replica, one

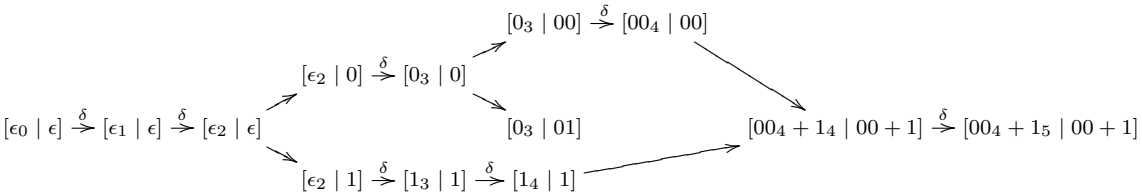


Figure 5.3: A set of partially ordered events with dynamic map clocks.

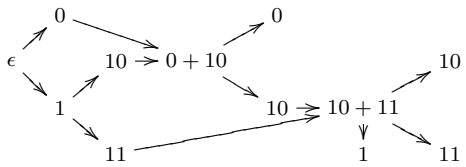


Figure 5.4: Non-pollution of the identity component in the dynamic map clock mechanism.

identity must be chosen among its exclusive identities and the associated counter must be incremented. This identity does not need to be the same for all updates in that replica and replicas can handover identities to other replicas.

As a consequence of these insights it is easy to conceive a scheme where replicas can fork by either specialising a binary identity (forking 010 would derive 0100 and 0101) or by partitioning controlled identities that were obtained upon joins (forking $010 + 10 + 111$ could derive $010 + 111$ and 10). This is the basic mechanism that supports dynamic map clocks and Figure 5.3 shows a run that illustrates this. More complex rules are enforced when handling joins and setting counters upon joins.

5.3.1 Non-Pollution of the Namespace

Considering the namespace pollution problem that was present on version stamps, it is easy to verify that dynamic map clocks are much more flexible on the handling of names. Figure 5.4 shows how the run that depicted a name pollution pattern in version stamps is easily handled by this mechanism.

In some way, dynamic map clocks try to ally the innovative management of identities that stems from version stamps with the benefits of classical counters

and their synthetic encoding of updates.

5.4 Discussion

Handling replication in highly decentralised systems and large scale settings—in number of nodes, geographical distance or communication latency between nodes—often implies the use of optimistic techniques in order to improve availability. In these scenarios, replicas are allowed to diverge from a consistent global state but reconciliation procedures and update propagation strategies are put in place so that consistency can eventually be restored. All these operations must rely on some dependency tracking mechanism in order to infer the causal relations between replica states. As mentioned before, the standard version vector mechanism assumes a consistent management of replicas names.

In decentralised distributed systems that face partitions, large membership changes under churn and disconnected operation, one can no longer rely on the assumption that globally unique names are available. A way of approaching these settings is to avoid determinism altogether and rely on probabilistic approaches such as generating random replica names, and assume some risk of name collisions [Heu59], or using sets of hashes of replica state to detect updates [KWK03] once again assuming some risk of collision. If reliability cannot be compromised, as is often the case, only a deterministic approach is appropriate. Determinism can only be obtained by recursive generation of names, the approach developed and formalised in our previous work on version stamps. However, as we have shown in the present chapter, recursive generation of names can easily introduce important growth problems in the space consumed by the version stamp mechanism.

With dynamic map clocks we achieve a better handling of the data space by avoiding unnecessary partitions of identifiers and concentrating on the important property that each replica must at a given moment have exclusive access to at least one globally unique identifier. Unlike version stamps, that do not use counters, dynamic map clocks are, in a sense, a hybrid mechanism that also relies on counters for registering updates. This usage of counters leads to important savings in size.

In addition, although the examples here have only shown runs with join operations, dynamic map clocks allow the use of messages when sending metadata and support unidirectional updating of dependency information.

While the theory of dynamic map clocks is still under development and a proper formalisation and formal proof is ongoing work, we already have a running implementation of the mechanism. This implementation has been tested on long random runs under various numbers of replicas and always evaluated as correct. This evaluation is done by contrasting the causal pre-order that the mechanism derives with the equivalent pre-order derived by causal histories. Causal histories are implemented by assuming global knowledge and adding unique update events to a set of events in each replica and relating this sets by set inclusion (see [ABF02b, SM94] for more details on causal histories). We can comment, from our experience, that incorrect mechanisms typically fail these checks after a small number of steps and do not stay correct in long random runs.

Another important property of dynamic map clocks, not present in version stamps, is their potential use as substitutes for vector clocks in autonomous decentralised settings. Vector clocks, that are at the core of causal message delivery protocols and distributed debugging, also rely on globally unique names thus facing the same problems of version vectors.

Chapter 6

Interval Tree Clocks: A Logical Clock for Dynamic Systems

Causality tracking mechanisms, such as vector clocks and version vectors, rely on mappings from globally unique identifiers to integer counters. In a system with a well known set of entities these ids can be pre-configured and given distinct positions in a vector or distinct names in a mapping. Id management is more problematic in dynamic systems, with large and highly variable number of entities, being worsened when network partitions occur. Present solutions for causality tracking are not appropriate to these increasingly common scenarios. In this chapter we introduce *Interval Tree Clocks*, a novel causality tracking mechanism that can be used in scenarios with a dynamic number of entities, allowing a completely decentralised creation of processes/replicas without need for global identifiers or global coordination. The mechanism has a variable size representation that adapts automatically to the number of existing entities, growing or shrinking appropriately. The representation is so compact that the mechanism can even be considered for scenarios with a fixed number of entities, which makes it a general substitute for vector clocks and version vectors.

6.1 Introduction

Ever since causality was introduced in distributed systems [Lam78], it has played an important role in the modelling of distributed computations. In the absence of global clocks, causality remains as a means to reason about the order of distributed events. In order to be useful, causality is implemented by concrete mechanisms, such as Vector Clocks [Fid89, Mat89c] and Version Vectors [PPR+83], where a compressed representation of the sets of events observed by processes or replicas is kept.

These mechanisms are based on a mapping from a globally unique identifier to an integer counter, so that each entity (i.e. process or replica) keeps track of how many events it knows from each other entity. A special and common case is when the number of entities is known: here ids can be integers, and a vector of counters can be used.

Nowadays, distributed systems are much less static and predictable than those traditionally considered when the basic causality tracking mechanisms were created. In dynamic distributed systems [MRT+05], the number of active entities varies during the system execution and in some settings, such as in peer-to-peer deployments, the level of change, due to churn, can be extremely high.

Causality tracking in dynamic settings is not new [Fid91] and several proposals analysed the dynamic creation and retirement of entities [RRP97, Gol98, PSTT96, Lan07, ABF02b]. However, in most cases localised retirement is not supported: all active entities must agree before an id can be removed [RRP97, Gol98, PSTT96] and a single unreachable entity will stall garbage collection. Localised retirement is only partially supported in [Lan07], while [ABF02b] has full support but the mechanism itself exhibits an unreasonable structural growth that its practical use is compromised [ABF07].

This chapter addresses causality tracking in dynamic settings and introduces Interval Tree Clocks (ITC), a novel causality tracking mechanism that generalises both Version Vectors and Vector Clocks. It does not require global ids but is able to create, retire and reuse them autonomously, with no need for global coordination; any entity can fork a new one and the number of entities can be reduced by joining

arbitrary pairs of entities; stamps tend to grow or shrink, adapting to the dynamic nature of the system. Contrary to some previous approaches, ITC is suitable for practical uses, as the space requirement scales well with the number of entities and grows modestly over time.

In the next section we review the related work. Section 6.3 introduces a model based on **fork**, event and **join** operations that factors out a kernel for the description of causality systems. Section 6.4 builds on the identified core operations and introduces a general framework that expresses the properties that must be met by concrete causality tracking mechanisms. Section 6.5 introduces the ITC mechanism and correctness argument under the framework. Before conclusions, in Section 6.7, we present in Section 6.6 a simple simulation based assessment of the space requirements of the mechanism.

6.2 Related Work

After Lamport's description of causality in distributed system [Lam78], subsequent work introduced the basic mechanisms and theory [PPR+83, Fid89, Mat89c, CB91]. We refer the interested reader to the survey in [SM94] and to the historical notes in [BR02]. After an initial focus on message passing systems, recent developments have improved causality tracking for replicated data: they addressed efficient coding for groups of related objects [MT05]; bounded representation of version vectors [AAB04]; and the semantics of reconciliation [GKK+06].

Fidge introduces in [Fid91] a model with a variable number of process ids. In this model process ids are assumed globally unique and are gradually introduced by process spawning events. No garbage collection of ids is performed when processes terminate.

Garbage collection of terminated ids requires additional metadata in order to assess that all active entities already witnessed the termination; otherwise, ids cannot be safely removed from the vectors. This approach is used in [Gol98, RRP97] together with the assumption of globally unique ids. In [PSTT96] the assumption of global ids is dropped and each entity is able to produce a globally unique id from

local information. A typical weakness in these systems is twofold: terminated ids cannot be reused; and garbage collection is hampered by even a single unreachable entity. In addition, when garbage collection cannot terminate, the associated metadata overhead cannot be freed. Since this overhead is substantial, when the likelihood of non termination is high, it can be more efficient not to garbage collect and keep the inactive ids.

The mechanism described in [Lan07] provides local retirement of ids but only for restricted termination patterns (a process can only be retired by joining a direct ancestor); moreover, the use of global ids is required.

Our own work in [ABF02b] introduced localised creation and retirements of ids and presented Version Stamps, a dynamic substitute to version vectors. Although still of theoretical interest as it does not use counters, and although it inspired the id management technique used in ITC, the technique was later found out to exhibit very adverse growth in common scenarios [ABF07]. The id management technique used in version stamps shares many properties with credit management techniques in termination detection algorithms [Mat89a, Hua89].

In order to control version vector growth, in Dynamo [DHJ+07] old inactive entries are garbage collected. Although the authors tune it so that in production systems errors are unlikely to be introduced, in general this can lead to resurgence of old updates. Mechanisms like ITC may help in avoiding the need for these aggressive pruning solutions.

6.3 Fork-Event-Join Model

Causality tracking mechanisms can be modelled by a set of core operations: **fork**, event and **join**, that act on stamps (logical clocks) whose structure is a pair (i, e), formed by an id and an event component that encodes causally known events. Fidge used in [Fid91] a model that bears some resemblance, although not making explicit the id component.

Causality is characterised by a partial order over the event components, $(E, \leq)$. In version vectors, this order is the pointwise order on the event component: $e \leq e'$

iff $\forall k.\ e[k] \leq e'[k]$. In causal histories [SM94], where event components are sets of event ids, the order is defined by set inclusion.

fork The **fork** operation allows the cloning of the causal past of a stamp, resulting in a pair of stamps that have identical copies of the event component and distinct ids; $\mathbf{fork}(i, e) = ((i_1, e), (i_2, e))$ such that $i_2 \neq i_1$. Typically, $i = i_1$ and i_2 is a new id. In some systems i_2 is obtained from an external source of unique ids, e.g. MAC addresses. In contrast, in Bayou [PSTT96] i_2 is a function of the original stamp $f((i, e))$; consecutive forks are assigned distinct ids since an event is issued to increment a counter after each **fork**.

peek A special case of **fork** when it is enough to obtain an *anonymous* stamp $(\mathbf{0}, e)$, with "null" identity, than can be used to transmit causal information but cannot register events, $\text{peek}((i, e)) = ((i, e), (\mathbf{0}, e))$. Anonymous stamps are typically used to create messages or as inactive copies for later debugging of distributed executions.

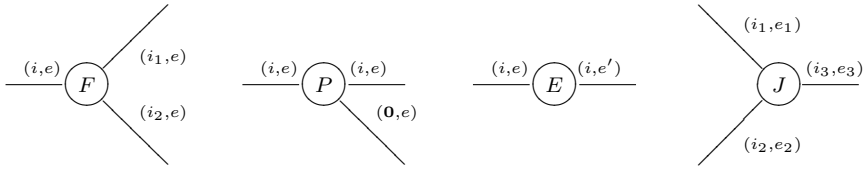
event An event operation adds a new event to the event component, so that if (i, e') results from $\text{event}((i, e))$ the causal ordering is such that $e < e'$. This action does a strict advance in the partial order such that e' is not dominated by any other entity and does not dominate more events than needed: for any other event component x in the system, $e' \not\leq x$ and when $x < e'$ then $x \leq e$. In version vectors the event operation increments a counter associated to the identity in the stamp: $\forall k \neq i.\ e'[k] = e[k]$ and $e'[i] = e[i] + 1$.

join This operation merges two stamps, producing a new one. If $\mathbf{join}((i_1, e_1), (i_2, e_2)) = (i_3, e_3)$, the resulting event component e_3 should be such that $e_1 \leq e_3$ and $e_2 \leq e_3$. Also, e_3 should not dominate more that either e_1 and e_2 did. This is obtained by the order theoretical join, $e_3 = e_1 \sqcup e_2$, that must be defined for all pairs; i.e. the order must form a join semilattice. In causal histories the join is defined by set union, and in version vectors it is obtained by the pointwise maximum of the two vectors.

The identity should be based on the provided ones, $i_3 = f(i_1, i_2)$ and kept globally unique (with the exception of anonymous ids). In most systems this

is obtained by keeping only one of the ids, but if ids are to be reused it should depend upon and incorporate both [ABF02b].

When one stamp is anonymous, **join** can also model message reception, where $\mathbf{join}((i, e_1), (\mathbf{0}, e_2)) = (i, e_1 \sqcup e_2)$. When both ids are defined, the **join** can be used to terminate an entity and collect its causal past. Also notice that joins can be applied when both stamps are anonymous, modelling in-transit aggregation of messages.



(a) fork (b) peek (c) event (d) **join**

Figure 6.1: Core operations.

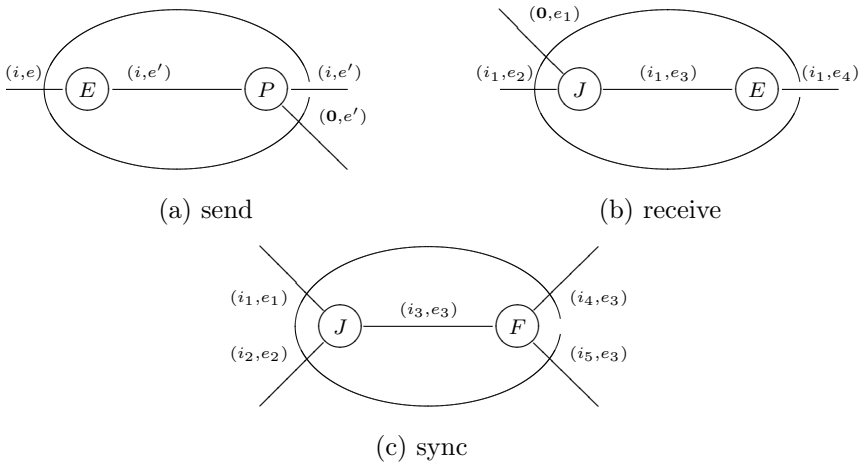
Classic operations can be described as a composition of these core operations:

send This operation is the atomic composition of event followed by peek. E.g. in vector clock systems, message sending is modelled by incrementing the local counter and then creating a new message.

receive A receive is the atomic composition of **join** followed by event. E.g. in vector clocks taking the pointwise maximum is followed by an increment of the local counter.

sync A sync is the atomic composition of **join** followed by **fork**. E.g. In version vector systems and in bounded version vectors [AAB04] it models the atomic synchronisation of two replicas.

Figure 6.2 depicts graphical representations of these composite operations, but other composite operations could also be easily described using the same set of core operations. For instance, a message multicast could be modelled as the atomic composition of an event operation followed by a sequence of peek operations.



(a) send (b) receive

(c) sync

Figure 6.2: Some composite operations.

Traditional descriptions assume a starting number of entities. This can be simulated by starting from an initial *seed* stamp and forking several times until the required number of entities is reached.

6.4 Function Space Based Clock Mechanisms

In this section we present a general framework which can be used to explain and instantiate concrete causality tracking mechanisms, such as our own ITC presented in the next section. Here stamps are described in terms of functions and some invariants are presented towards ensuring correctness. Actual mechanisms can be seen as finite encodings of such functions. Correctness of each mechanism will follow directly from the correctness of the encoding and from respecting the corresponding semantics and conditions to be met by each operation. In the following we will make use of the standard pointwise sum, product, scaling, partial ordering and join of functions:

$$(f + g)(x) \doteq f(x) + g(x),$$

$$(f \cdot g)(x) \doteq f(x) \cdot g(x),$$

$$(n \cdot g)(x) \doteq n \cdot g(x),$$

$$f \leq g \doteq \forall x.\ f(x) \leq g(x),$$

$$(f \sqcup g)(x) \doteq f(x) \sqcup g(x),$$

and of a function $\mathbf{0}$ that maps all elements to 0:

$$\mathbf{0} \doteq \lambda x.\, 0.$$

A stamp will consist of a pair (i, e): the identity and the event components, both functions from some arbitrary domain to natural numbers. The identity component is a characteristic function (maps elements to $\{0, 1\}$) that defines the set of elements in the domain available to *inflate* ("increment") the event function when an event occurs. We chose to use the characteristic function instead of the set as it leads to better notation. The essential point towards ensuring a correct tracking of causality is to be able to inflate the mapping of some element which no other entity (process or replica) has access to[1]. This means each entity having an identity which maps to 1 some element which is mapped to 0 in all other entities. This is expressed by the following invariant over the identity components of all entities:

$$\forall i.\ (i \cdot \bigsqcup_{i' \neq i} i') \neq i.$$

We adopt a less general but more useful invariant, as it can be maintained by local operations without access to global knowledge. It consists of having disjointness of the parts of the domain that are mapped to 1 in each entity; i.e. non-overlapping graphs for any pair of id functions.

$$\forall i_1 \neq i_2.\ i_1 \cdot i_2 = \mathbf{0}.$$

Comparison of stamps is made through the event component:

$$(i_1, e_1) \leq (i_2, e_2) \doteq e_1 \leq e_2.$$

[1] If this property is not met it can still be possible to form an order that is compatible with causality, but where some concurrent events appear as ordered. This is the case in Lamport clocks [Lam78] and in plausible clocks [TRA99] where the stated invariant does not hold. A Lamport clock can be modelled by having the same identity in all entities.

Join takes two stamps, and returns a stamp that causally dominates both (therefore, the event component is a join of the event components), and has the elements from both identities available for future event accounting:

$$\mathbf{join}((i_1, e_1), (i_2, e_2)) \doteq (i_1 + i_2, e_1 \sqcup e_2).$$

Fork can be any function that takes a stamp and returns two stamps which keep the same event component, but split between them the available elements in the identity; i.e. any function:

$$\mathbf{fork}((i, e)) \doteq ((i_1, e), (i_2, e)) \qquad \text{subject to } i_1 + i_2 = i \text{ and } i_1 \cdot i_2 = \mathbf{0}.$$

Peek is a special case of fork, which results in one *anonymous* stamp with $\mathbf{0}$ identity and another which keeps all the elements in the identity to itself:

$$\text{peek}((i, e)) \doteq ((i, e), (\mathbf{0}, e)).$$

Event can be any function that takes a stamp and returns another with the same identity and with an event component inflated on any arbitrary set of elements available in the identity:

$$\text{event}((i, e)) = (i, e + f \cdot i) \qquad \text{for any } f \text{ such that } f \cdot i > \mathbf{0}.$$

An event cannot be applied to an anonymous stamp as no element in the domain is available to be inflated.

6.5 Interval Tree Clocks

We now describe *Interval Tree Clocks*, a novel clock mechanism that can be used in scenarios with a dynamic number of entities, allowing a completely decentralised creation of processes/replicas without need for global identifiers. The mechanism has a variable size representation that adapts automatically to the number of existing entities, growing or shrinking appropriately. There are two essential differences

between ITC and classic clock mechanisms, from the point of view of our function space framework:

- in classic mechanisms each entity uses a fixed, pre-defined function for id; in ITC the id component of entities is manipulated to adapt to the dynamic number of entities;
- classic mechanisms are based on functions over a discrete and typically finite domain; ITC is based on functions over a continuous infinite domain ($\mathbb{R}$) with emphasis on the interval $[0, 1)$; this domain can be split into an arbitrary number of subintervals as needed.

The idea is that each entity has available, in the id, a set of intervals that it can use to *inflate* the event component and to give to the successors when forking; a join operation joins the sets of intervals. Each interval results from successive partitions of $[0, 1)$ into equal subintervals; the set of intervals is described by a binary tree structure. Another binary tree structure is also used for the event component, but this time to describe a mapping of intervals to integers. To describe the mechanism in terms of functions, it is useful to define a *unit pulse* function[2]:

$$\mathbf{1} \doteq \lambda x. \begin{cases} 1 & x \geq 0 \wedge x < 1, \\ 0 & x < 0 \vee x \geq 1. \end{cases}$$

The id component is an *id tree* with the recursive form (where i, i_1, i_2 range over id trees):

$$i \doteq 0 \mid 1 \mid (i_1, i_2).$$

We define a semantic function for the interpretation of id trees as functions:

$$\begin{aligned} [\![0]\!] &= \mathbf{0} \\ [\![1]\!] &= \mathbf{1} \\ [\![(i1, i2)]\!] &= \lambda x. [\![i_1]\!](2x) + [\![i_2]\!](2x - 1). \end{aligned}$$

[2]In this chapter we use the *lambda calculus* notation for defining unary functions: a function is anonymously defined by a lambda expression which expresses its action on its argument. For instance, the "increment" function f such that $f(x) = x + 1$ would be expressed as $\lambda x. x + 1$

These functions can be 1 for some subintervals of $[0,1)$ and 0 otherwise. For an id (i_1, i_2), the functions corresponding to the two subtrees are transformed so as to be non-zero in two non-overlapping subintervals: i_1 in the interval $[0, 1/2)$ and i_2 in the interval $[1/2, 1)$. As an example, $(1,(0,1))$ represents the function $\lambda x.\,\mathbf{1}(2x) + (\lambda x.\,\mathbf{1}(2x-1))(2x-1)$. We will also use a graphical notation, which is based on the graph of the function over $[0,1)$. Examples:

$$(1,(0,1)) \quad \sim$$

$$((0,(1,0)),(1,0)) \quad \sim$$

The event component is a binary *event tree* with non-negative integers in nodes; using e, e_1, e_2 to range over event trees and n over non-negative integers:

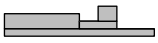
$$e \doteq n \mid (n, e_1, e_2).$$

We define a semantic function for the interpretation of these trees as functions:

$$\begin{aligned} [\![n]\!] &= n \cdot \mathbf{1} \\ [\![(n, e1, e2)]\!] &= n \cdot \mathbf{1} + \lambda x.\,[\![e_1]\!](2x) + [\![e_2]\!](2x-1). \end{aligned}$$

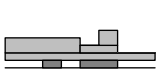
This means that the value for an element in some subinterval is the sum of a *base* value, common for the whole interval, plus a *relative* value from the corresponding subtree. We will also use a graphical notation for the event component; again, it is based on the graph of the function, obtained by "stacking" the corresponding parts. An example:

$$(1,2,(0,(1,0,2),0)) \quad \sim$$



A stamp in ITC is a pair (i, e), where i is an id tree and e an event tree; we will also use a graphical notation based on stacking the two components:

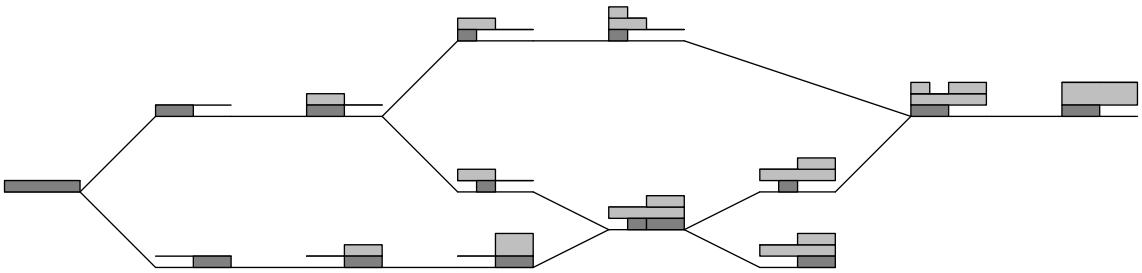
$$(((0,(1,0)),(1,0)),(1,2,(0,(1,0,2),0))) \quad \sim$$



ITC makes use what we call the *seed* stamp, $(1, 0)$, from which we can fork as desired to obtain an initial configuration.

6.5.1 An Example

We now present an example to illustrate the intuition behind the mechanism, showing a run with a dynamic number of entities in the fork-event-join model. The run starts by a single entity, with the *seed* stamp, which forks into two; one of these suffers one event and forks; the other suffers two events. At this point there are three entities. Then, one entity suffers an event while the remaining two synchronise by doing a join followed by a fork.



The example shows how ITC adapts to the number of entities and allows simplifications to occur upon joins or events. While the first two forks had to split a node in the id tree, the third one makes use of the two available subtrees. The final join leads to a simplification in the id by merging two subtrees. It can be seen that each event always inflates the event tree in intervals available in the id. The event after the final join managed to perform an inflation in a way such that the resulting event function is represented by a single integer.

6.5.2 Normal Form

There can be several equivalent representations for a given function. ITC is conceived so as to keep stamps in a *normal form*, for the representations of both id and event functions. This is important not only for having compact representations but also to allow simple definitions of the operations on stamps (fork, event,

join) as shown below. As an example, for the unit pulse, we have:

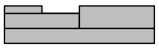
$$\mathbf{1} \sim 1 \equiv (1,1) \equiv (1,(1,1)) \equiv ((1,1),1) \equiv \ldots$$

This means that, if after a join the resulting id is $(1,(1,1))$, we can simplify it to 1. Normalisation of the id component can be obtained by applying the following function when building the id tree recursively:

$$\begin{aligned} \mathrm{norm}((0,0)) &= 0, \\ \mathrm{norm}((1,1)) &= 1, \\ \mathrm{norm}(i) &= i. \end{aligned}$$

The event component can be also normalised, preserving its interpretation as a function. Two examples:

$(2,1,1) \sim$  $\equiv$  $\sim 3,$

$(2,(2,1,0),3) \sim$  $\equiv$  $\sim (4,(0,1,0),1).$

To normalise the event component we will make use of the following operators to "lift" or "sink" a tree:

$$\begin{aligned} \lfloor n \rfloor m &= n+m, \\ \lfloor (n,e_1,e_2) \rfloor m &= (n+m,e_1,e_2), \\ n{\downarrow_m} &= n-m, \\ (n,e_1,e_2){\downarrow_m} &= (n-m,e_1,e_2). \end{aligned}$$

Normalisation of the event component can be obtained by applying the following function when building a tree recursively (where m and n range over integers

and e_1 and e_2 over normalised event trees) :

$$\begin{aligned} \text{norm}(n) &= n, \\ \text{norm}((n, m, m)) &= n + m, \\ \text{norm}((n, e_1, e_2)) &= (n + m, e_1{\downarrow_m}, e_2{\downarrow_m}), \text{ where } m = \min(\min(e_1), \min(e_2)), \end{aligned}$$

where min applied to a tree returns the minimum value of the corresponding function in the range $[0, 1)$:

$$\min(e) = \min_{x \in [0,1)} [\![e]\!](x),$$

which can be obtained by the recursive function over event trees:

$$\begin{aligned} \min(n) &= n, \\ \min((n, e_1, e_2)) &= n + \min(\min(e_1), \min(e_2)), \end{aligned}$$

or more simply, assuming the event tree is normalised:

$$\begin{aligned} \min(n) &= n, \\ \min((n, e_1, e_2)) &= n, \end{aligned}$$

which explores the property that in a normalised event tree, one of the subtrees has minimum equal to 0. We will also make use of the analogous max function over event trees that returns the maximum value of the corresponding function in the range $[0, 1)$, and can be obtained by the recursive function:

$$\begin{aligned} \max(n) &= n, \\ \max((n, e_1, e_2)) &= n + \max(\max(e_1), \max(e_2)). \end{aligned}$$

6.5.3 Operations over ITC

We now present the operations on ITC for the fork-event-join model. They are defined so as to respect the operations and invariants from the function space

based framework presented in the previous section. All the functions below take as input and give as result stamps in the normal form.

Comparison

Comparison of ITC can be derived from the pointwise comparison of the corresponding functions:

$$(i_1, e_1) \leq (i_2, e_2) \doteq [\![e_1]\!] \leq [\![e_2]\!].$$

It is trivial to see that this can be computed through a recursive function over normalised event trees; i.e. $(i_1, e_1) \leq (i_2, e_2) \Leftrightarrow \text{leq}(e_1, e_2)$, with leq defined as (where l and r range over the "left" and "right" subtrees):

$$\begin{aligned}
\text{leq}(n_1, n_2) &= n_1 \leq n_2, \\
\text{leq}(n_1, (n_2, l_2, r_2)) &= n_1 \leq n_2, \\
\text{leq}((n_1, l_1, r_1), n_2) &= n_1 \leq n_2 \wedge \text{leq}(\lfloor l_1 \rfloor n_1, n_2) \wedge \text{leq}(\lfloor r_1 \rfloor n_1, n_2), \\
\text{leq}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= n_1 \leq n_2 \wedge \text{leq}(\lfloor l_1 \rfloor n_1, \lfloor l_2 \rfloor n_2) \wedge \text{leq}(\lfloor r_1 \rfloor n_1, \lfloor r_2 \rfloor n_2).
\end{aligned}$$

Fork

Forking preserves the event component, and must split the id in two parts whose corresponding functions do not overlap and give the original one when added.

$$\textbf{fork}(i, e) \doteq ((i_1, e), (i_2, e)), \text{ where } (i_1, i_2) = \text{split}(i),$$

for a function split such that:

$$(i_1, i_2) = \text{split}(i) \Rightarrow [\![i_1]\!] \times [\![i_2]\!] = \mathbf{0} \wedge [\![i_1]\!] + [\![i_2]\!] = [\![i]\!].$$

This is satisfied naturally using the following recursive function over id trees, as the two subtrees of an id component always represent functions that do not overlap:

$$\begin{aligned}
\text{split}(0) &= (0,0),\\
\text{split}(1) &= ((1,0),(0,1)),\\
\text{split}((0,i)) &= ((0,i_1),(0,i_2)),\ \text{where } (i_1,i_2) = \text{split}(i),\\
\text{split}((i,0)) &= ((i_1,0),(i_2,0)),\ \text{where } (i_1,i_2) = \text{split}(i),\\
\text{split}((i_1,i_2)) &= ((i_1,0),(0,i_2))
\end{aligned}$$

Join

Joining two entities is made by summing the corresponding id functions and making a join of the corresponding event functions:

$$\text{join}((i_1,e_1),(i_2,e_2)) \doteq (\text{sum}(i_1,i_2),\text{join}(e_1,e_2)),$$

for a sum function over identities and a join function over event trees such that:

$$\begin{aligned}
[\![\text{sum}(i_1,i_2)]\!] &= [\![i_1]\!] + [\![i_2]\!],\\
[\![\text{join}(e_1,e_2)]\!] &= [\![e_1]\!] \sqcup [\![e_2]\!].
\end{aligned}$$

The sum function that respects the above condition and also produces a normalised id is:

$$\begin{aligned}
\text{sum}(0,i) &= i,\\
\text{sum}(i,0) &= i,\\
\text{sum}((l_1,r_1),(l_2,r_2)) &= \text{norm}((\text{sum}(l_1,l_2),\text{sum}(r_1,r_2))).
\end{aligned}$$

Likewise, the join function over event trees, producing a normalised event tree

is:

$$\begin{aligned}
\text{join}(n_1, n_2) &= \max(n_1, n_2), \\
\text{join}(n_1, (n_2, l_2, r_2)) &= \text{join}((n_1, 0, 0), (n_2, l_2, r_2)), \\
\text{join}((n_1, l_1, r_1), n_2) &= \text{join}((n_1, l_1, r_1), (n_2, 0, 0)), \\
\text{join}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= \text{join}((n_2, l_2, r_2), (n_1, l_1, r_1)), \text{ if } n_1 > n_2, \\
\text{join}((n_1, l_1, r_1), (n_2, l_2, r_2)) &= \text{norm}((n_1, \text{join}(l_1, \lfloor l_2 \rfloor n_2 - n_1), \text{join}(r_1, \lfloor r_2 \rfloor n_2 - n_1))).
\end{aligned}$$

Event

The event operation is substantially more complex than the others. While fork and join have a simple natural definition, event has a larger freedom of implementation while respecting the condition:

$\text{event}((i, e)) = (i, e')$, subject to $[\![e']\!] = [\![e]\!] + f \cdot [\![i]\!]$ for any f such that $f \cdot [\![i]\!] > \mathbf{0}$.

Event cannot be applied to anonymous stamps; it has the precondition that the id is non-null; i.e. $i \neq 0$. We can use any subset of the available id to inflate the event function. The freedom of which part to inflate is explored in ITC so as to simplify the event tree. Considering the final event in our larger example:

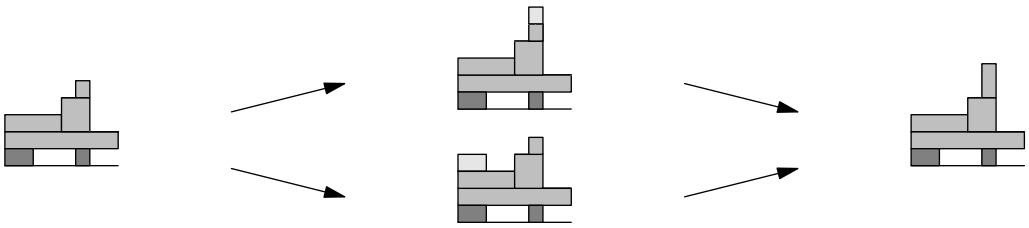
The event operation was able to fill the missing part in a tree so as to allow its simplification to a single integer. In general, the event operation can use several parts of the id, and may simplify several subtrees simultaneously. The operation performs all simplifications in the event tree that are possible given the id tree. If some simplification is possible (which means the corresponding function was inflated), the resulting tree is returned; otherwise another procedure is applied, that "grows" some subtree, preferably only incrementing an integer if possible. The event operation is defined resorting to these two functions (fill and grow) defined below:

$$\text{event}(i,e) = \begin{cases} (i, \text{fill}(i,e)) & \text{if } \text{fill}(i,e) \neq e, \\ (i, e') & \text{otherwise, where } (e', c) = \text{grow}(i,e). \end{cases}$$

Fill either succeeds in doing one or more simplifications, or returns an unmodified tree; it never increments an integer that would not lead to simplifying the tree:

$$\begin{aligned} \text{fill}(0,e) &= e, \\ \text{fill}(1,e) &= \max(e), \\ \text{fill}(i,n) &= n, \\ \text{fill}((1,i_r),(n,e_l,e_r)) &= \text{norm}((n, \max(\max(e_l), \min(e'_r)), e'_r)), \\ &\quad \text{where } e'_r = \text{fill}(i_r, e_r), \\ \text{fill}((i_l,1),(n,e_l,e_r)) &= \text{norm}((n, e'_l, \max(\max(e_r), \min(e'_l)))), \\ &\quad \text{where } e'_l = \text{fill}(i_l, e_l), \\ \text{fill}((i_l,i_r),(n,e_l,e_r)) &= \text{norm}((n, \text{fill}(i_l,e_l), \text{fill}(i_r,e_r))). \end{aligned}$$

In the following example, fill is unable to perform any simplification and grow is used. From the two candidate inflations shown in light grey, the one chosen requires a simple integer increment, while the other would require expanding a node:



Grow performs a dynamic programming based optimisation to choose the inflation that can be performed, given the available id tree, so as to minimise the cost of the event tree growth. It is defined recursively, returning the new event tree and cost, so that:

- incrementing an integer is preferable over expanding an integer to a tuple;

- to disambiguate, an operation near the root is preferable to one farther away.

$$
\begin{aligned}
\text{grow}(1, n) &= (n+1, 0), \\
\text{grow}(i, n) &= (e', c+N), \text{ where } (e', c) = \text{grow}(i, (n, 0, 0)), \\
&\quad \text{and } N \text{ is some large constant}, \\
\text{grow}((0, i_r), (n, e_l, e_r)) &= ((n, e_l, e'_r), c_r + 1), \text{ where } (e'_r, c_r) = \text{grow}(i_r, e_r), \\
\text{grow}((i_l, 0), (n, e_l, e_r)) &= ((n, e'_l, e_r), c_l + 1), \text{ where } (e'_l, c_l) = \text{grow}(i_l, e_l), \\
\text{grow}((i_l, i_r), (n, e_l, e_r)) &= \begin{cases} ((n, e'_l, e_r), c_l + 1) & \text{if } c_l < c_r, \\ ((n, e_l, e'_r), c_r + 1) & \text{if } c_l \geq c_r, \end{cases} \\
&\quad \text{where } (e'_l, c_l) = \text{grow}(i_l, e_l) \text{ and } (e'_r, c_r) = \text{grow}(i_r, e_r).
\end{aligned}
$$

The definition makes use of a constant N that should be greater than the maximum tree depth that arises. This is a practical choice, to have the cost as a simple integer. We could avoid it by having the cost as a pair under lexicographic order, but it would "pollute" the presentation and be a distracting element.

6.6 Exercising ITCs

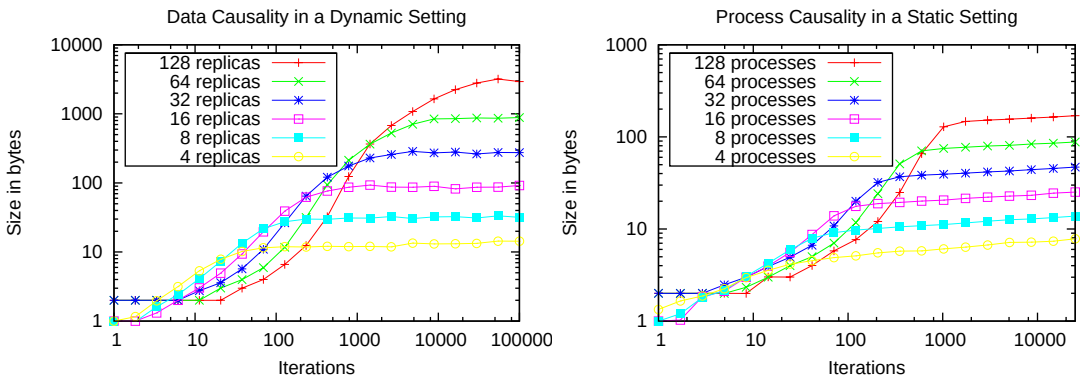


Figure 6.3: Average space consumption of an ITC stamp, in dynamic and static settings.

In order to have a rough insight of ITC space consumption we exercised its usage for both dynamic and static scenarios, using a mix of data and process causality. For data causality in dynamic scenarios, each iteration consists of forking, recording an event and joining two replicas, each performed on random replicas, leading to constantly evolving ids. This pattern maintains the number of existing replicas while exercising id management under churn. For process causality in a static scenario, we operate on a fixed set of processes doing message exchanges (via peek and join) and recording internal events; here ids remain unchanged, since messages are anonymous.

The charts in Figure 6.3 depict the mean size (using the binary encoding shown in Appendix 6.8) of an ITC across 100 runs of 25,000 iterations for process causality and 100,000 iterations for data causality and for different numbers of active entities (pre-created by forking a *seed* stamp before iterating). It shows that space consumption basically stabilises after a number of iterations. These results show that ITCs can in fact be used as a practical mechanism for data and process causality in dynamic systems, contrary to Version Stamps [ABF02b] that have storage cost growing unreasonably over time.

In order to put these numbers in perspective, the Microsoft Windows operating system [PTR+07] uses 128 bits Universally Unique Identifiers (UUIDs) and 32 bit counters. The storage cost of a version vector for 128 replicas would be 2560 bytes using a mapping from ids to counters and 512 bytes using a vector. The mean size of an ITC for this scenario (at the end of the iterations) would be less than 2900 bytes for dynamic scenarios and slightly above 170 bytes for static ones. While vectors can be represented in a more compact way (e.g. factoring out the smallest number), such optimisations would be irrelevant for dynamic scenarios, where most of the cost stems from the UUIDs.

6.7 Conclusions

We have introduced Interval Tree Clocks, a novel logical clock mechanism for dynamic systems, where processes/replicas can be created or retired in a decentralised

fashion. The mechanism has been presented using a model (fork-event-join) that can serve as a kernel to describe all classic operations (like message sending, symmetric synchronisation and process creation/retirement), being suitable for both process and data causality scenarios.

We have presented a general framework for clock mechanisms, where stamps can be seen as finite representations of a pair of functions over a continuous domain; the event component serves to perform comparison or join (performed pointwise); the identity component defines a set of intervals where the event component can be inflated (a generalisation of the classic counter increment). ITC is a concrete mechanism that instantiates the framework, using trees to describe functions on sets of intervals. The framework opens the way for research on future alternative mechanisms that use different representations of functions.

Previous approaches to causality tracking for dynamic systems either require access to globally unique ids; do not reuse ids of retired entities; require global coordination for garbage collection of ids; or exhibit an intolerable growth in terms of space consumption (our previous approach). ITC is the first mechanism for dynamic systems that avoids all these problems, can be used for both process and data causality, and requires a modest space consumption, making it a general purpose mechanism, even for static systems.

6.8 A Binary Encoding for ITC

Here we describe a compact encoding of ITC as strings of bits. It may be relevant when stamp size is an issue, e.g. when many entities are involved; it is appropriate to being transmitted or stored persistently as a single *blob*. We do not attempt to present an optimal (in some way) encoding, but a sensible one, which was used in the space consumption analysis.

As an event tree tends to have very few large numbers near the root and many very small numbers at the leaves; this prompts a variable length representation for integers, where small integers occupy just a few bits. Also, common cases like trees with only the left or right subtree, or with 0 for the base value are treated

as special cases.

We use a notation (inspired by the bit syntax from the Erlang programming language) where: $\ll x, y, z \gg$ is a string of bits resulting from concatenating x, y and z; and $n{:}b$ represents number n encoded in b bits. An example: $\ll 2{:}3, 0{:}1, 1{:}2 \gg$ represents the string of 6 bits 010001.

$$
\begin{aligned}
enc((i,e)) &= \ll enc_i(i), enc_e(e) \gg .
\end{aligned}
$$

$$
\begin{aligned}
enc_i(0) &= \ll 0{:}2, 0{:}1 \gg , \\
enc_i(1) &= \ll 0{:}2, 1{:}1 \gg , \\
enc_i((0,i)) &= \ll 1{:}2, enc_i(i) \gg , \\
enc_i((i,0)) &= \ll 2{:}2, enc_i(i) \gg , \\
enc_i((i_l,i_r)) &= \ll 3{:}2, enc_i(i_l), enc_i(i_r) \gg .
\end{aligned}
$$

$$
\begin{aligned}
enc_e((0,0,e_r)) &= \ll 0{:}1, 0{:}2, enc_e(e_r) \gg , \\
enc_e((0,e_l,0)) &= \ll 0{:}1, 1{:}2, enc_e(e_l) \gg , \\
enc_e((0,e_l,e_r)) &= \ll 0{:}1, 2{:}2, enc_e(e_l), enc_e(e_r) \gg , \\
enc_e((n,0,e_r)) &= \ll 0{:}1, 3{:}2, 0{:}1, 0{:}1, enc_e(n), enc_e(e_r) \gg , \\
enc_e((n,e_l,0)) &= \ll 0{:}1, 3{:}2, 0{:}1, 1{:}1, enc_e(n), enc_e(e_l) \gg , \\
enc_e((n,e_l,e_r)) &= \ll 0{:}1, 3{:}2, 1{:}1, enc_e(n), enc_e(e_l), enc_e(e_r) \gg , \\
enc_e(n) &= \ll 1{:}1, enc_n(n,2) \gg .
\end{aligned}
$$

$$
enc_n(n,B) = \begin{cases} \ll 0{:}1, n{:}B \gg & \text{if } n < 2^B, \\ \ll 1{:}1, enc_n(n-2^B, B+1) \gg & \text{otherwise.} \end{cases}
$$

Chapter 7

Conclusions

This thesis explored the extension of causality tracking mechanisms to dynamic settings, which nowadays are increasingly predominant. The initial motivation was driven by the desire to build an ad-hoc replication system, where individual files can be autonomously replicated and merged. This motivation in systems design helped to give some context to the more theoretical work that was to come. The first mechanism developed fitted the ad-hoc replication setting and had the interesting capability of coalescing the causality tracking stamps if all replicas merged. It also helped to realize that some mechanisms that could be made to operate only on the frontier elements of a consistent cut, as was the case, would not be feasible for tracking causality across the whole computation (e.g. for distributed debugging).

Interestingly, it was a deficiency of DVS that lead to the mechanism that would overcome these limitations. Since DVS would not scale nicely under specific runs it become apparent that the cause for such behaviour was an invariant relating the id and event components that forced, when doing a fork, the forking of each bitstring in an id, even if several strings were available in the id. If id buddies did not meet before being forked again, the growth could be uncontrolled. This made the mechanism, although theoretically interesting and unique as it did not use counters, of less practical relevance. If this invariant was removed, id management would become much simpler as it would suffice to ensure that each active entity

has access to an exclusive id. By combining counters with the same id struture it became possible to reuse ids more agressively; it also became possible to track the computation past (and not only co-existing elements in a consistent cut).

The first version of such a mechanism, Dynamic Map Clocks, was thus a combination of DVS style ids (with weaker restrictions) and integer counters. Obtaining a valid combination was not trivial, since some subtle properties had to be met. The exploratory research consisted in building candidate mechanisms and testing them in lengthy runs that compared them to causal histories until no inconsistencies were found. A working implementation was obtained but the mechanism was never completely formalized; the flat structure of ids lead to unelegant definitions of the operations and consequently difficult formal proof.

An important step was going from a flat structure to the hierarchical tree-like structure adopted for Interval Tree Clocks. This lead to elegant definitions and allowed more simplifications of stamps. Describing the mechanism in terms of function spaces was important towards ensuring correctness.

The use of trees in the encoding of ITC also highlighted the possibility of making counters relative to the parent node (and not absolute as in the flat structure of DMC). This leads to a representation where, as information is gathered and the system evolves in a run, counter values flow from the leaves towards the root. This results in small counter values in the leaves, and fewer large numbers as we approach the root. It was then natural to explore this property in a variable size binary encoding and achieve significant space savings.

The function space representation and the causality kernel, devised together with ITC were important steps in the understanding of causality. In particular, function spaces help the understanding in terms of area inclusion of the relations between causal pasts. Identities can now be seen as keys that allow access to a given portion of the area where exclusive growth is allowed for a given active entity. It makes it easy to accept that keys can be freely exchanged and that different owners can, in succession, take possession of any given key.

If keys are shared (identities overlap) we will not longer have a characterization of causality but an order that potentially orders more events, and that is only

consistent with causality. This is what occurs in scalar clocks and in plausibly clocks; these mechanism can be modeled with function spaces and seen not to respect the invariants needed to characterize causality.

7.1 Summary of Contributions

The main contributions of this thesis can be summarized as the following:

- Autonomous Identity Management. Autonomous creation of ids for causality tracking was already explored in Bayou [PSTT96]; however, although ids could be terminated there was no provision for id reuse. The technique used in this thesis is fairly simple and similar ones have possibly been used in many contexts, but to our knowledge this was the first consistent use in causality tracking settings.

- Dynamic Version Stamps. This mechanism represents the only deterministic technique (in contrast to Hash Histories) that does not use counters and that can track data causality in a dynamic system (even if restricted to tracking the frontier elements of a consistent cut). The fact that causality can be tracked without using counters or a direct enumeration of the whole causal history is a new finding.

- FEJ Model. The Fork-Event-Join model helps to understand the relations among existing applications of causality. It contains the core operations that are enough to model different scenarios that are tpypically described resorting to different models. Standard operations (like message send, receive, broadcast or symetric synchronization) can be modelled as an atomic composition of the core operations.

- Function Spaces. This modeling of causal history was crucial to the understanding of ITC, but also constitutes a generic tool for understanding other causality tracking mechanisms. It can be used in the process of developing future mechanisms aiming to improve on ITC.

- Interval Tree Clocks. This mechanism, the more recent in this line of research, promises to represent the first viable mechanism for tracking causality in dynamic systems, while solving most the problems that classic mechanism such as version vectors or vector clocks suffer under such scenarios. The mechanism is more complex that the classic ones, but if suitable reference implementations are provided, adoption by the systems community can be facilitated.

7.2 Research Directions

7.2.1 Assessing ITC Load

Storage consumption of ITC depends on several factors: whether new entities keep being created, communication patterns, and activity rates among entities. The exploratory simulation in Chapter 6 already shows how space consumption evolves under some synthetic executions.

Further empirical analysis of space consumption and computational complexity should be addressed for synthetic runs involving broadcasts. It is expected that this will show a positive impact on space consumption since, with broadcasts, information is spread more quickly and more opportunities for tree simplification are presented. Further analysis can be made using trace driven simulations from diverse application scenarios: group communication, databases, filesystems, CSCW, and version control systems.

7.2.2 Behaviour under Churn

An important class of application scenarios involves systems that are either subject to churn or have a large membership of entities albeit with a few entities responsible for most activity [DHJ+07]. In these scenarios, there is an expressive pollution of ids that are either inactive or present a very low activity. It is thus relevant to better access the behaviour of ITC in these settings and verify if they compare favorably to classic mechanisms. Preliminary explorations with synthetic runs give

some confidence on this possibility.

7.2.3 Identity Theft

A more pro-active solution in the control of stamp growth in the presence of churn or low activity entities, is to explore the possibility of having buddies taking over ids of entities with low activity or suspected to have failed. Its seems possible to devise an age detection algorithm that tries to detect ids that have not communicated updates for some time.

When ids are suspected to be inactive, its id buddy can deterministically take over the ids in order to enable future simplifications. This opens the possibility of loosing updates in the suspected inactive ids, but such a possibility can be a reasonable tradeoff since the resulting relation will still be consistent with causality (i.e. a new kind of plausible clock).

Bibliography

[AAB04] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. Bounded version vectors. In Guerraoui [Gue04], pages 102–116.

[ABF00] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Panasync: Dependency tracking among file copies. In Paulo Guedes, editor, *Ninth ACM SIGOPS European Workshop*, pages 7–12. DIKU - University of Copenhagen, 2000.

[ABF02a] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps – decentralized version vectors. Technical Report UMDITR2002.01, Universidade do Minho, DI/CCTC, 2002.

[ABF02b] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps – decentralized version vectors. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 544–551. IEEE Computer Society, 2002.

[ABF07] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Improving on version stamps. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, OTM Academy Doctoral Consortium, MONET, OnToContent, ORM, PerSys, PPN, RDDS, SSWS, and SWWS 2007, Vilamoura, Portugal, November 25–30, 2007, Proceedings, Part II*, volume 4806 of *Lecture Notes in Computer Science*, pages 1025–1031. Springer, 2007.

[AR07] James Aspnes and Eric Ruppert. An introduction to population protocols. *Bulletin of the European Association for Theoretical Computer Science*, 93:98–117, October 2007. Columns: Distributed Computing.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[BA99] Carlos Baquero and Paulo Sérgio Almeida. Towards efficient time-stamping for autonomous versioning. In *Actas informais do EPCM'99, Encontro Português de Computação Nómada*, 1999.

[BCCS98] Maria Butrico, Henry Chang, Norman Cohen, and Dennis G. Shea. Data synchronization in mobile network computer – reference specification. In *WMR'98, ECOOP'98 Workshop Reader*. Springer Verlag, 1998.

[BM99] Carlos Baquero and Francisco Moura. Causality in autonomous mobile systems. In *Third European Research Seminar on Advances in Distributed Systems*. Broadcast, EPFL-LSE, April 1999.

[BR02] Roberto Baldoni and Michel Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2), 2002.

[CB89] Bernadette Charron-Bost. Combinatorics and geometry of consistent cuts: Application to concurrency theory. In *WDAG: International Workshop on Distributed Algorithms*. LNCS, Springer-Verlag, 1989.

[CB91] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, 1991.

[CBDGF95] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. Program. Lang. Syst.*, 17(1):157–179, 1995.

[CES04] David Culler, Deborah Estrin, and Mani Srivastava. Guest Editors' introduction: Overview of sensor networks. *Computer*, 37(8), August 2004.

[CL85] Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *Transactions on Computer Systems*, 3(1):63–75, 1985.

[CM91] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Workshop on Parallel and Distributed Debugging*, pages 167–174, 1991.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007.

[Fid89] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 55–66, 1989.

[Fid91] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

[GHM+90] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus Replicated File System. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[GKK+06] Michael B. Greenwald, Sanjeev Khanna, Keshav Kunal, Benjamin C. Pierce, and Alan Schmitt. Agreeing to agree: Conflict resolution for optimistically replicated data. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2006.

[Gol98] Richard G. Golden, III. Efficient vector time with dynamic process creation and termination. *Journal of Parallel and Distributed Computing (JPDC)*, 55(1):109–120, 1998.

[GRR+98] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER Workshops*, pages 254–265, 1998.

[Gue04] Rachid Guerraoui, editor. *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4–7, 2004, Proceedings*, volume 3274 of *Lecture Notes in Computer Science*. Springer, 2004.

[Heu59] Gerald A. Heuer. Estimation in a certain probability problem. *The American Mathematical Monthly*, 66(8):704–706, 1959.

[HHRB92] Peter Honeyman, Larry Huston, Jim Rees, and Dave Bachmann. The LITTLE WORK Project. In *In Proceedings of the Third Workshop on Workstation Operating Systems*, pages 11–14, Key Biscayne, Florida, U.S., 1992. IEEE Computer Society Press.

[HNI+98] Jaap Haartsen, Mahmoud Naghshineh, Jon Inouye, Olaf Joeressen, and Warren Allen. Bluetooth: Vision, goals, and architecture. *ACM Mobile Computing and Communications Review*, 2(4):38–45, October 1998.

[HRMB03] Jean-Michel Hélary, Michel Raynal, Giovanna Melideo, and Roberto Baldoni. Efficient causality-tracking timestamping. *IEEE Transactions on Knowledge and Data Engineering*, 15, 2003.

[Hua89] Shing-Tsaan Huang. Detecting termination of distributed computations by external agents. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS)*, pages 79–84, Washington, DC, 1989. IEEE Computer Society.

[HW88] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences (21st HICSS'88)*, pages 166–175, Kailua-Kona, HI, January 1988. IEEE.

[Kno65] Kenneth C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–625, 1965.

[KPR94] Geoffrey H. Kuenning, Gerald J. Popek, and Peter L. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *In USENIX Conference Proceedings*, pages 291–303, 1994.

[KS91] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, US, 1991.

[KWK03] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23nd International Conference on Distributed Computing Systems (ICDCS)*, pages 670–677. IEEE Computer Society, 2003.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lan07] Tobias Landes. Tree clocks: An efficient and entirely dynamic logical time system. In Helmar Burkhart, editor, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, as part of the 25th IASTED International Multi-Conference on Applied Informatics, February 13–15 2007, Innsbruck, Austria*, pages 349–354. IASTED/ACTA Press, 2007.

[Mat87] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[Mat89a] Friedemann Mattern. Global quiescence detection based on credit distribution and recovery. *IPL: Information Processing Letters*, 30, 1989.

[Mat89b] Friedemann Mattern. *Verteilte Basisalgorithmen*, volume 226 of *Informatik-Fachberichte*. Springer-Verlag, 1989.

[Mat89c] Friedemann Mattern. Virtual time and global clocks in distributed systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[MJK+00] Robert Morris, John Jannoti, Frans Kaashoek, Jinyang Li, and Douglas Decouto. Carnet: A scalable ad hoc wireless network system. In Paulo Guedes, editor, *Ninth ACM SIGOPS European Workshop*, pages 61–65. DIKU - University of Copenhagen, 2000.

[MN91] Marzullo and Neiger. Detection of global state predicates. In *WDAG: International Workshop on Distributed Algorithms*. LNCS, Springer-Verlag, 1991.

[MRT+05] Achour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr El Abbadi. From static distributed systems to dynamic systems. In *Proceedings 24th IEEE Symposium on Reliable Distributed Systems (24th SRDS'05)*, pages 109–118, Orlando, FL, USA, October 2005. IEEE Computer Society.

[MT05] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in winfs. In Pierre Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

[Ora01] Andy Oram, editor. *Peer-to-peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, Sebastopol, California, 2001.

[PPR+83] Douglas Stott Parker, Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David A. Edwards, Stephen Kiser, and Charles S. Kline. Detection of mutual inconsistency in distributed systems. *Transactions on Software Engineering*, 9(3):240–247, 1983.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly

consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, 1997.

[PSTT96] Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. Bayou: Replicated database services for world-wide applications. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996. http://www.parc.xerox.com/csl/projects/bayou/.

[PTR+07] Daniel Peek, Douglas B. Terry, Venugopalan Ramasubramanian, Meg Walraed-Sullivan, Thomas L. Rodeheffer, and Ted Wobber. Fast encounter-based synchronization for mobile devices. *Digital Information Management, 2007. ICDIM '07. 2nd International Conference on*, 2:750–755, 2007.

[Ray06] Michel Raynal. From static distributed systems to dynamic systems: an approach for a first step. In *ICDCS Workshops*. IEEE Computer Society, 2006.

[RPG+96] Peter Reiher, Jerry Popek, Michial Gunter, John Salomone, and David Ratner. Peer-to-peer reconciliation based replication for mobile computers. In Max Muhlhauser, editor, *Special Issues in Object-Oriented Programming, ECOOP'96 II Workshop on Mobility and Replication*. Dpunkt Verlag, 1996.

[RPR96] David Ratner, Gerald J. Popek, and Peter Reiher. The ward model: A scalable replication architecture for mobility. In *In Workshop on Object Replication and Mobile Computing*, 1996.

[RRP97] David Ratner, Peter Reiher, and Gerald Popek. Dynamic version vector maintenance. Technical Report CSD-970022, Department of Computer Science, University of California, Los Angeles, 1997.

[RRP99] David Ratner, Peter Reiher, and Gerald J. Popeky. Roam: A scalable replication system for mobile computing. In *In Workshop on Mobile Databases and Distributed Systems (MDDS)*, pages 96–104, 1999.

[Sat96] Mahadev Satyanarayanan. Fundamental challenges in mobile computing. In *PODC*, pages 1–7, 1996.

[SK92] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, August 1992.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SM94] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the Holy Grail. *Distributed Computing*, 3(7):149–174, 1994.

[Smy78] Michael B. Smyth. Power domains. *Journal of Computer and System Sciences*, 16(1):23–36, February 1978.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[TRA99] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–196, 1999.

[TTP+95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado*, 1995.

[Val93] Céline Valot. Characterizing the accuracy of distributed timestamps. *SIGPLAN Notices*, 28(12):43–52, 1993.

[Vog08] Werner Vogels. Beyond server consolidation. *ACM Queue: Tomorrow's Computing Today*, 6(1):20–26, January 2008.

[WB84] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *PODC*, pages 233–242, 1984.

[WRB99] An-I Wang, Peter L. Reiher, and Rajive Bagrodia. A simulation evaluation of optimistic replicated filing in mobile environments. In *IPCCC*, pages 43–51. IEEE, 1999.

[YV01] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *Symposium on Operating Systems Principles*, pages 29–42, 2001.

Appendix A

Version Stamps Reference Implementation

A.1 Core Implementation

```
1  # vim: set ts=4 sw=4 ai et:
2
3  from auxiliary import *
4
5  optimize = 1
6
7  debug = True
8
9  class VersionStamp(object):
10
11     ch_counter = 0
12     id_map = {}
13
14
15     def __init__(self):
16         self.id = []
17         self.up = []
18         self.ch = set()
19
20
21     def __repr__(self):
22         return "(%s, %s)" % (repr(self.id), repr(self.up))
23
24
25     def fork(self):
```

```
26
27          # Create a new VS
28          new = VersionStamp()
29
30          # Split the identity set
31          if len(self.id) == 0:
32              new.id = ['1']
33              self.id = ['0']
34          else:
35              new.id = [ f + '1' for f in self.id ]
36              self.id = [ f + '0' for f in self.id ]
37
38          # Copy the update set
39          new.up = self.up[:]
40
41          # Copy the causal history
42          new.ch = self.ch.copy()
43
44          # Return the new VS
45          return new
46
47
48      def join(self, other):
49
50          # Join the identity and update components
51          self.id = self.id + [ e for e in other.id if e not in self.id ]
52          self.up = self.up + [ e for e in other.up if e not in self.up ]
53
54          id = self.id
55          up = self.up
56
57          # Coalesce identity
58          stop = 0
59          while not stop:
60              s = [ e[:-1] for e in id if e[:-1] + str(1 - int(e[-1])) in id ]
61              for e in list(set(s)): # unique(s)
62                  id.remove(e + '0')
63                  id.remove(e + '1')
64                  if len(e) > 0: id.append(e)
65              stop = len(s) == 0
66
67          # Remove unnecessary detail from the update component
68          for e in up[:]:
69              s = [ f for f in id if f != e and sleq(f, e) ]
70              if s:
71                  up.remove(e)
72                  if s[0] not in up:
73                      up.append(s[0])
74
75          self.id.sort()
```

```
76          self.up.sort()
77
78          # Make sure up is an antichain
79          for u in up[:]:
80              if [ f for f in up if f != u and sleq(u, f) ]:
81                  up.remove(u)
82
83          for u in self.up:
84              assert len([ i for i in self.id if sleq(u, i) ]) >= 1
85
86          # Join the causal histories
87          self.ch = self.ch.union(other.ch)
88
89          # Destroy the other VS
90          other.id = []
91          other.up = []
92          other.ch.clear()
93
94
95      def update(self):
96
97          self.up = self.id[:]
98
99          # Add a token to the causal history
100         VersionStamp.ch_counter = VersionStamp.ch_counter + 1
101         self.ch.add(self.ch_counter)
102
103         # Record the update in the debugging identity map
104         self.id_map[self.ch_counter - 1] = repr(self.up)
105
106
107     event = update
108
109
110     def leq(self, other):
111         for i in self.up:
112             if [j for j in other.up if sleq(i, j)] == []:
113                 return False
114         return True
115
116
117     __le__ = leq
118
119
120     def ch_leq(self, other):
121         return self.ch <= other.ch
```

A.2 Auxiliary Functions

```
1 # vim: set ts=4 sw=4 et ai:
2
3 def sleq(s1, s2):
4     return s2.startswith(s1)
```

Appendix B

Dynamic Map Clocks Reference Implementation

B.1 Core Implementation

```
1  # vim: set ts=4 sw=4 ai et:
2
3  from auxiliary import *
4
5  debug = True
6
7  class DynamicMapClock(object):
8
9      ch_counter = 0
10     id_map = {}
11
12
13     def __init__(self):
14         self.id = set([''])
15         self.up = {}
16         self.ch = set()
17
18
19     def __repr__(self):
20         return "(%s, %s)" % (repr(self.id), repr(self.up))
21
22
23     def fork(self):
24
25         def split_identity():
```

```
26              """
27              Split the identity set in half, regardless of the length
28              of the identity strings or their relation to the update
29              set. When the identity set has one string only then it
30              must be split, each resulting fragment having its own
31              suffix ('0' or '1').
32              """
33
34              # Split identity strings.
35              oid = list(self.id)
36              nid = oid[len(oid) / 2:]
37              oid = oid[:len(oid) / 2]
38
39              # If there is a single identity string it must be split.
40              if not oid:
41                  bs = nid.pop()
42                  oid.append(bs + '0')
43                  nid.append(bs + '1')
44
45              return set(oid), set(nid)
46
47          if debug: assert is_idantichain(self.id)
48
49          # Create a new DMC
50          new = DynamicMapClock()
51
52          # Split the identity set
53          self.id, new.id = split_identity()
54          if debug: assert is_idantichain(self.id)
55
56          # Copy the update set
57          new.up = self.up.copy()
58
59          # Copy the causal history
60          new.ch = self.ch.copy()
61
62          # Return the new DMC
63          return new
64
65
66      def join(self, other):
67
68          def coalesce_identity():
69              """
70              Coalesce complementary (sibling) identity strings.
71              """
72
73              stop = False
74              while not stop:
75                  stop = True
```

```
76                  for i in self.id:
77                      if i[:-1]+'0' in self.id and i[:-1]+'1' in self.id:
78                          self.id.remove(i[:-1]+'0')
79                          self.id.remove(i[:-1]+'1')
80                          self.id.add(i[:-1])
81                          stop = False
82                          break
83
84          if debug: assert is_idantichain(self.id)
85
86          # Join the identity sets
87          self.id = self.id.union(other.id)
88          if debug: assert is_idantichain(self.id)
89
90          # Coalesce the identity set (non-essential for correctness)
91          coalesce_identity()
92          if debug: assert is_idantichain(self.id)
93
94          # Join the update sets maxing the counter values of duplicate
95          # identity strings.
96          for i in other.up:
97              self.up[i] = max(other.up[i], self.up.get(i, 0))
98
99          # Remove dominated fragments from the update set
100         mk_upantichain(self.up)
101         if debug: assert is_upantichain(self.up)
102
103         # Join the causal histories
104         self.ch = self.ch.union(other.ch)
105
106         # Destroy the other DMC
107         other.id.clear()
108         other.up.clear()
109         other.ch.clear()
110
111
112     def update(self):
113
114         def choose_identity():
115             """
116             Choose an identity string in order to register an update.
117             Any identity string will do, but several strategies can be
118             used in order no minimize space.
119             """
120
121             same_ids = self.id.intersection(self.up)
122             inline_ids = set([i for i in self.id for j in self.up
123                 if i != j and is_idinline(i, j)])
124             # alternative strategies
125             if len(same_ids) != 0:
```

```
126                 i = same_ids.pop()
127             elif len(inline_ids) != 0:
128                 i = inline_ids.pop()
129             else:
130                 i = self.id.copy().pop()     # default strategy
131             return i
132 
133         if debug: assert is_idantichain(self.id)
134 
135         # Choose an identity string
136         i = choose_identity()
137 
138         # Update or add a fragment to the update set
139         counters = [self.up[j] for j in self.up if is_idinline(i, j)] # B1
140         #counters = [self.up[j] for j in self.up if sleq(i, j)] # B2
141         self.up[i] = max(counters + [0]) + 1
142 
143         # Remove dominated fragments from the update set
144         mk_upantichain(self.up)
145         if debug: assert is_upantichain(self.up)
146 
147         # Add a token to the causal history
148         DynamicMapClock.ch_counter = DynamicMapClock.ch_counter + 1
149         self.ch.add(self.ch_counter)
150 
151         # Record the update in the debugging identity map
152         #self.id_map[self.ch_counter - 1] = i + " " + str(self.up[i])
153 
154 
155     event = update
156 
157 
158     def sync(self, other):
159 
160         if debug: assert is_idantichain(self.id)
161 
162         # Join the update sets maxing the counter values of duplicate
163         # identity strings.
164         for i in other.up:
165             self.up[i] = max(other.up[i], self.up.get(i, 0))
166 
167         # Remove dominated fragments from the update set
168         mk_upantichain(self.up)
169         if debug: assert is_upantichain(self.up)
170 
171         # Join the causal histories
172         self.ch = self.ch.union(other.ch)
173 
174 
175     def leq(self, other):
```

```
176             for i in self.up.items():
177                 if [j for j in other.up.items() if snleq(i, j)] == []:
178                     return False
179             return True
180 
181 
182         __le__ = leq
183 
184 
185         def ch_leq(self, other):
186             return self.ch <= other.ch
```

B.2 Auxiliary Functions

```
1  # vim: set ts=4 sw=4 et ai:
2  
3  def sleq(s1, s2):
4      return s2.startswith(s1)
5  
6  def snleq(sn1, sn2):
7      return sleq(sn1[0], sn2[0]) and sn1[1] <= sn2[1] # A1
8  
9  def is_idinline(s1, s2):
10     return sleq(s1, s2) or sleq(s2, s1)
11 
12 def is_upinline(sn1, sn2):
13     return snleq(sn1, sn2) or snleq(sn2, sn1)
14 
15 def is_idantichain(id):
16     return [i for i in id for j in id
17         if i != j and is_idinline(i, j)] == []
18 
19 def is_upantichain(up):
20     return [i for i in up.items() for j in up.items()
21         if i != j and is_upinline(i, j)] == []
22 
23 def mk_upantichain(up):
24     for i in up.items():
25         if [j for j in up.items() if i != j and snleq(i, j)] != []:
26             del up[i[0]]
```

Appendix C

Interval Tree Clocks Reference Implementation

C.1 Core Implementation

```
1  -module(dmct).
2  -export([new/0, event/1, join/2, fork/1, peek/1, leq/2]).
3  -export([len/1, str/1, enc/1]).
4  -compile({inline, [{min,2}, {max,2}, {drop,2}, {lift,2}, {base,1},
5          {height,1}]}).
6
7  new() -> {1, 0}.
8
9  join({I1, E1}, {I2, E2}) -> {sum(I1,I2), join_ev(E1, E2)}.
10
11 fork({I, E}) ->
12  {I1, I2} = split(I),
13  {{I1, E}, {I2, E}}.
14
15 peek({I, E}) -> {{0, E}, {I, E}}.
16
17 event({I, E}) ->
18  {I,
19   case fill(I, E) of
20     E -> {_, E1} = grow(I, E), E1;
21     E1 -> E1
22   end
23  }.
24
25 leq({_, E1}, {_, E2}) -> leq_ev(E1, E2).
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

leq_ev({N1, L1, R1}, {N2, L2, R2}) ->
 N1 =< N2 andalso
 leq_ev(lift(N1, L1), lift(N2, L2)) andalso
 leq_ev(lift(N1, R1), lift(N2, R2));

leq_ev({N1, L1, R1}, N2) ->
 N1 =< N2 andalso
 leq_ev(lift(N1, L1), N2) andalso
 leq_ev(lift(N1, R1), N2);

leq_ev(N1, {N2, _, _}) -> N1 =< N2;

leq_ev(N1, N2) -> N1 =< N2.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Normal form

norm_id({0, 0}) -> 0;
norm_id({1, 1}) -> 1;
norm_id(X) -> X.

norm_ev({N, M, M}) when is_integer(M) -> N + M;
norm_ev({N, L, R}) ->
 M = min(base(L), base(R)),
 {N + M, drop(M, L), drop(M, R)}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

sum(0, X) -> X;
sum(X, 0) -> X;
sum({L1,R1}, {L2, R2}) -> norm_id({sum(L1, L2), sum(R1, R2)}).

split(0) -> {0, 0};
split(1) -> {{1, 0}, {0, 1}};
split({0, I}) -> {I1, I2} = split(I), {{0, I1}, {0, I2}};
split({I, 0}) -> {I1, I2} = split(I), {{I1, 0}, {I2, 0}};
split({I1, I2}) -> {{I1, 0}, {0, I2}}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

join_ev(E1={N1, _, _}, E2={N2, _, _}) when N1 > N2 -> join_ev(E2, E1);
join_ev({N1, L1, R1}, {N2, L2, R2}) when N1 =< N2 ->
 D = N2 - N1,
 norm_ev({N1, join_ev(L1, lift(D, L2)), join_ev(R1, lift(D, R2))});
join_ev(N1, {N2, L2, R2}) -> join_ev({N1, 0, 0}, {N2, L2, R2});
join_ev({N1, L1, R1}, N2) -> join_ev({N1, L1, R1}, {N2, 0, 0});
join_ev(N1, N2) -> max(N1, N2).
```

```

fill(0, E) -> E;
fill(1, E={_, _, _}) -> height(E);
fill(_, N) when is_integer(N) -> N;
fill({1, R}, {N, El, Er}) ->
 Er1 = fill(R, Er),
 D = max(height(El), base(Er1)),
 norm_ev({N, D, Er1});
fill({L, 1}, {N, El, Er}) ->
 El1 = fill(L, El),
 D = max(height(Er), base(El1)),
 norm_ev({N, El1, D});
fill({L, R}, {N, El, Er}) ->
 norm_ev({N, fill(L, El), fill(R, Er)}).

grow(1, N) when is_integer(N)->
 {0, N + 1};
grow({0, I}, {N, L, R}) ->
 {H, E1} = grow(I, R),
 {H + 1, {N, L, E1}};
grow({I, 0}, {N, L, R}) ->
 {H, E1} = grow(I, L),
 {H + 1, {N, E1, R}};
grow({Il, Ir}, {N, L, R}) ->
 {Hl, El} = grow(Il, L),
 {Hr, Er} = grow(Ir, R),
 if
   Hl < Hr -> {Hl + 1, {N, El, R}};
   true ->    {Hr + 1, {N, L, Er}}
 end;
grow(I, N) when is_integer(N)->
 {H, E} = grow(I, {N, 0, 0}),
 {H + 1000, E}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

height({N, L, R}) -> N + max(height(L), height(R));
height(N) -> N.

base({N, _, _}) -> N;
base(N) -> N.

lift(M, {N, L, R}) -> {N + M, L ,R};
lift(M, N) -> N + M.

drop(M, {N, L, R}) when M =< N -> {N - M, L ,R};
drop(M, N) when M =< N -> N - M.

max(X, Y) when X =< Y -> Y;
max(X, _) -> X.
```

```

min(X, Y) when X =< Y -> X;
min(_, Y) -> Y.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

enc({I, E}) -> << (enci(I))/bits, (ence(E))/bits >>.

enci(0) -> <<0:2, 0:1>>;
enci(1) -> <<0:2, 1:1>>;
enci({0, I}) -> <<1:2, (enci(I))/bits>>;
enci({I, 0}) -> <<2:2, (enci(I))/bits>>;
enci({L, R}) -> <<3:2, (enci(L))/bits, (enci(R))/bits>>.

ence({0, 0, R}) -> <<0:1, 0:2, (ence(R))/bits>>;
ence({0, L, 0}) -> <<0:1, 1:2, (ence(L))/bits>>;
ence({0, L, R}) -> <<0:1, 2:2, (ence(L))/bits, (ence(R))/bits>>;
ence({N, 0, R}) -> <<0:1, 3:2, 0:1, 0:1,
                     (ence(N))/bits, (ence(R))/bits>>;
ence({N, L, 0}) -> <<0:1, 3:2, 0:1, 1:1,
                     (ence(N))/bits, (ence(L))/bits>>;
ence({N, L, R}) -> <<0:1, 3:2, 1:1,
                     (ence(N))/bits, (ence(L))/bits, (ence(R))/bits>>;
ence(N) -> encn(N, 2, <<1:1>>).

encn(N, B, Acc) when N < (1 bsl B) -> <<Acc/bits, 0:1, N:B>>;
encn(N, B, Acc) -> encn(N - (1 bsl B), B + 1, <<Acc/bits, 1:1>>).

len(D) -> size(enc(D)).

str({I, E}) -> [lists:flatten(stri(I)), lists:flatten(stre(E))].

stri(0) -> "0";
stri(1) -> "";
stri({0, I}) -> "R"++stri(I);
stri({I, 0}) -> "L"++stri(I);
stri({L, R}) -> ["(L"++stri(L), "+", "R"++stri(R), ")"].

stre({N, L, 0}) -> [stre(N), "L", stre(L)];
stre({N, 0, R}) -> [stre(N), "R", stre(R)];
stre({N, L, R}) -> [stre(N), "(L", stre(L), "+R", stre(R), ")"];
stre(N) when N > 0 -> integer_to_list(N);
stre(_) -> "".
```